

# Universal Serial Bus (USB)

Article06/26/2024

This reference section describes the driver programming interfaces that are included in the [Windows Driver Kit \(WDK\)](#). The programming interfaces are used for developing drivers that interact with USB devices, host controllers, and connectors. These interfaces include export functions that the drivers can call, callback routines that the driver can implement, I/O requests that the driver can send to the Microsoft-provided USB driver stack, and various data structures that are used in those requests.

For the programming guide, see [Universal Serial Bus \(USB\)](#).

## Common USB client driver reference

A Windows Driver Model (WDM)-based USB client driver can call functions to communicate with the Microsoft-provided USB driver stack. These functions are defined in Usbdlib.h and the client driver requires the Usbdex.lib library. The library gets loaded and statically linked to the client driver module when built. A client driver that calls these routines can run on Windows Vista and later versions of Windows.

## Programming Guide

[Developing Windows client drivers for USB devices.](#)

## Headers

- [usb.h](#)
- [usbbusif.h](#)
- [usbdlib.h](#)
- [usbfnattach.h](#)
- [usbfnbase.h](#)
- [usbfioctl.h](#)
- [usbioclt.h](#)
- [usbspec.h](#)

## Deprecated functions, IOCTL requests for all USB drivers

These functions are deprecated.

| Do not use.

- USBD\_CalculateUsbBandwidth
- USBD\_CreateConfigurationRequest
- USBD\_Debug\_LogEntry
- USBD\_GetUSBDIVersion
- USBD\_ParseConfigurationDescriptor
- USBD\_QueryBusTime
- USBD\_RegisterHcFilter

These I/O requests are deprecated or reserved for internal use.

| USB client drivers must not use these I/O requests:

- IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_OFF
- IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_ON
- IOCTL\_USB\_DIAGNOSTIC\_MODE\_OFF
- IOCTL\_USB\_DIAGNOSTIC\_MODE\_ON
- IOCTL\_USB\_GET\_HUB\_CAPABILITIES
- IOCTL\_USB\_HCD\_DISABLE\_PORT
- IOCTL\_USB\_HCD\_ENABLE\_PORT
- IOCTL\_USB\_HCD\_GET\_STATS\_1
- IOCTL\_USB\_HCD\_GET\_STATS\_2
- IOCTL\_USB\_RESET\_HUB

## Kernel-Mode IOCTLs

USB client drivers can receive or send any of the following I/O requests in kernel mode:

- IOCTL\_INTERNAL\_USB\_CYCLE\_PORT
- IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO
- IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME
- IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO
- IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME
- IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS
- IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS
- IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE
- IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION
- IOCTL\_INTERNAL\_USB\_RESET\_PORT
- IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION
- IOCTL\_INTERNAL\_USB\_SUBMIT\_URB

- [IOCTL\\_INTERNAL\\_USB\\_UNREGISTER\\_COMPOSITE\\_DEVICE](#)

## User-Mode IOCTLs sent by applications and services

USB client drivers receive these user-mode I/O control requests at the kernel level:

- [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#)
- [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#)
- [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_NAME](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_INFORMATION](#)
- [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)
- [IOCTL\\_USB\\_GET\\_ROOT\\_HUB\\_NAME](#)
- [IOCTL\\_USB\\_HUB\\_CYCLE\\_PORT](#)

## Dual-role controller driver reference

A USB driver for a dual-role controller can behave as a host controller or a function controller depending on the hardware. Dual-role controllers are common on mobile devices and allow for connections to PCs, as well as USB peripherals like keyboards and mice. A mobile device can behave as a peripheral when connected to a PC, allowing you to transfer files between your PC and the mobile device. In that scenario, the controller on the device operates in the function role. Conversely, the controller can operate in the host role when connected to USB peripherals like storage drives, keyboard, mice.

One of the main responsibilities of a driver for a dual-role controller is to switch between those two roles, tearing down the previous role's device node and loading the device node for the new role. When writing the driver, use the WDF class extension-client driver model. For more information about the WDF class extension-client driver model, see Ursdevice.h.

## Dual-role controller driver programming guide

For information about enabling a Windows system for USB dual-role support, see [USB Dual Role Driver Stack Architecture](#).

## Dual-role controller driver headers

- [ursdevice.h](#)
- [urstypes.h](#)

## Emulated host controller driver reference

Windows drivers can present non-USB devices as emulated USB devices. By using the WDF class extension-client driver model, you can write a driver that translates USB-level constructs (reset, data transfers) to the actual underlying bus by using the hardware's interface. The class extension and the client driver represent an emulated host controller with a root hub that is capable of presenting an attached device to the system as a USB device.

- USB device emulation class extension (UdeCx) is an in-box driver included Windows 10.
- The client driver written by an IHV/OEM and referred to as the UDE client driver.

The driver pair loads as the functional device object (FDO) in the host controller device stack. The UDE client driver communicates with Udecx by using a set of methods and event callback functions to handle device requests and notify the class extension about various events.

## Emulated host controller programming guide

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

## Emulated host controller headers

- [udecxurb.h](#)
- [udecxusbdevice.h](#)
- [udecxusbendpoint.h](#)
- [udecxwdfdevice.h](#)

## Function class driver reference

A USB function class driver implements the functionality of a specific group of interfaces on the USB device. The class driver handle requests issued by user mode services, or it can forwards requests to USB function class extension (UFX) and its function client driver. Certain class drivers are included in Windows, such as Media Transfer Protocol (MTP) and IpOverUsb. Windows also provides a generic kernel-mode class driver, GenericUSBFn.sys. If a particular interface or functionality isn't provided by a system-supplied driver, you might need to write a function class driver. You can implement the class driver as a kernel-mode driver by using Windows Driver Frameworks (WDF). Or you can implement it as a user-mode service. In that case, your class driver must be paired with the system-supplied class driver, GenericUSBFn.sys. For example, the MTP class driver runs as a user-mode service that transferring files to and from the device.

## Function class driver headers

- [usbfbase.h](#)
- [usbfioctl.h](#)

## USB function controller client driver reference

The USB function client driver is responsible for implementing function controller-specific operations. The client driver communicates with the USB function class extension (UFX) module to handle 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.

## USB function controller client driver programming guide

- [Write a USB function controller client driver](#)

## USB function controller client driver headers

- [ufxclient.h](#)

## Filter driver for supporting USB chargers

Write a filter driver that supports detection of chargers, if the function controller uses the in-box Synopsys and Chipidea drivers. If you're 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](#).

## Filter driver for supporting USB chargers programming guide

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

## Filter driver for supporting USB chargers headers

- [usbfnattach.h](#)
- [ufxbase.h](#)
- [ufxproprietarycharger.h](#)

## Host controller driver reference

The USB host controller extension is a system-supplied extension to the Kernel-Mode Driver Framework (KMDF). Within the Microsoft USB Driver Stack Architecture, USB host controller extension (UCX) provides functionality to assist a host controller client driver in managing a USB host controller device. The client driver handles hardware operations and events, power management, and PnP events. UCX serves as an abstracted interface to the rest of the Microsoft USB 3.0 stack, queues requests to the client driver, and performs other tasks.

If you're developing an xHCI host controller that isn't 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 the UCX class extension.

## Host controller driver programming guide

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

## Host controller driver headers

- [ucxclass.h](#)
- [ucxcontroller.h](#)
- [ucxendpoint.h](#)
- [ucxroothub.h](#)
- [ucxsstreams.h](#)
- [ucxusbdevice.h](#)

# Type-C driver reference

Windows 10 introduces support for the new USB connector: USB Type-C. You can write a driver for these scenarios:

[+] Expand table

Scenario	Headers	Programming Guide
If your USB Type-C hardware has the capability of handling the power delivery (PD) state machine.	<a href="#">ucmmanager.h</a>	<a href="#">Write a USB Type-C connector driver</a>
If your driver wants to participate in the policy decisions for USB Type-C connectors.	<a href="#">Usbpapi.h</a>	<a href="#">Write a USB Type-C Policy Manager client driver</a>
If your hardware doesn't support PD.	<a href="#">ucmtcpdevice.h</a> <a href="#">ucmtcpiglobals.h</a> <a href="#">ucmtcpportcontroller.h</a> <a href="#">ucmtcpportcontrollerrequests.h</a> <a href="#">ucmtypes.h</a>	<a href="#">Write a USB Type-C port controller driver.</a>
If your embedded controller is connected over non-ACPI transport	<a href="#">Ucmucsicx.h</a> <a href="#">Ucmucsdevice.h</a> <a href="#">Ucmucsfuncenum.h</a> <a href="#">Ucmucsglobals.h</a> <a href="#">Ucmucsippm.h</a> <a href="#">Ucmucsippmrequests.h</a> <a href="#">Ucmucsispec.h</a>	<a href="#">Write a UCSI client driver</a>

## IOCTLs

[+] Expand table

<a href="#">IOCTL_GET_HCD_DRIVERKEY_NAME</a>
The IOCTL_GET_HCD_DRIVERKEY_NAME I/O control request retrieves the driver key name in the registry for a USB host controller driver.
<a href="#">IOCTL_INTERNAL_USB_CYCLE_PORT</a>
The IOCTL_INTERNAL_USB_CYCLE_PORT I/O request simulates a device unplug and replug on the port associated with the PDO.

## [IOCTL\\_INTERNAL\\_USB\\_ENABLE\\_PORT](#)

The IOCTL\_INTERNAL\_USB\_ENABLE\_PORT IOCTL has been deprecated. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_BUS\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO I/O request queries the bus driver for certain bus information.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_BUSGUID\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_BUSGUID\_INFO IOCTL has been deprecated. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_CONTROLLER\\_NAME](#)

The IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME I/O request queries the bus driver for the device name of the USB host controller.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_CONFIG\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO I/O request returns information about a USB device and the hub it is attached to.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_HANDLE](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_HANDLE\\_EX](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE\_EX IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_HUB\\_COUNT](#)

The IOCTL\_INTERNAL\_USB\_GET\_HUB\_COUNT IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_HUB\\_NAME](#)

The IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME I/O request is used by drivers to retrieve the UNICODE symbolic name for the target PDO if the PDO is for a hub.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_PARENT\\_HUB\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_PARENT\_HUB\_INFO is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_PORT\\_STATUS](#)

The IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS I/O request queries the status of the PDO.

[IOCTL\\_INTERNAL\\_USB\\_GET\\_PORT\\_STATUS](#) is a kernel-mode I/O control request. This request targets the USB hub PDO. This IOCTL must be sent at IRQL = PASSIVE\_LEVEL.

#### [IOCTL\\_INTERNAL\\_USB\\_GET\\_ROOTHUB\\_PDO](#)

The [IOCTL\\_INTERNAL\\_USB\\_GET\\_ROOTHUB\\_PDO](#) IOCTL is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_GET\\_TOPOLOGY\\_ADDRESS](#)

The [IOCTL\\_INTERNAL\\_USB\\_GET\\_TOPOLOGY\\_ADDRESS](#) I/O request returns information about the host controller the USB device is attached to, and the device's location in the USB device tree.

#### [IOCTL\\_INTERNAL\\_USB\\_GET\\_TT\\_DEVICE\\_HANDLE](#)

The [IOCTL\\_INTERNAL\\_USB\\_GET\\_TT\\_DEVICE\\_HANDLE](#) is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_NOTIFY\\_IDLE\\_READY](#)

The [IOCTL\\_INTERNAL\\_USB\\_NOTIFY\\_IDLE\\_READY](#) IOCTL is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_RECORD\\_FAILURE](#)

The [IOCTL\\_INTERNAL\\_USB\\_RECORD\\_FAILURE](#) IOCTL is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

The [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#) I/O request registers the driver of a USB multi-function device (composite driver) with the underlying USB driver stack.

#### [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_RESUME](#)

The [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_RESUME](#) IOCTL is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_SUSPEND](#)

The [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_SUSPEND](#) IOCTL is used by the USB hub driver. Do not use.

#### [IOCTL\\_INTERNAL\\_USB\\_REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#)

The [IOCTL\\_INTERNAL\\_USB\\_REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#) I/O request is sent by the driver of a Universal Serial Bus (USB) multi-function device (composite driver) to request remote wake-up notifications from a specific function in the device.

#### [IOCTL\\_INTERNAL\\_USB\\_RESET\\_PORT](#)

The [IOCTL\\_INTERNAL\\_USB\\_RESET\\_PORT](#) I/O control request is used by a driver to reset the upstream port of the device it manages.

## [IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_IDLE\\_NOTIFICATION](#)

The IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION I/O request is used by drivers to inform the USB bus driver that a device is idle and can be suspended.

## [IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_URB](#)

The IOCTL\_INTERNAL\_USB\_SUBMIT\_URB I/O control request is used by drivers to submit an URB to the bus driver. IOCTL\_INTERNAL\_USB\_SUBMIT\_URB is a kernel-mode I/O control request. This request targets the USB hub PDO.

## [IOCTL\\_INTERNAL\\_USB\\_UNREGISTER\\_COMPOSITE\\_DEVICE](#)

The IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE I/O request unregisters the driver of a USB multi-function device (composite driver) and releases all resources that are associated with registration.

## [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#)

The USB class driver sends this request to activate the bus so that the driver can prepare to process bus events and handle traffic.

## [IOCTL\\_INTERNAL\\_USBFN\\_BUS\\_EVENT\\_NOTIFICATION](#)

The USB class driver sends this request to prepare for notifications received from the USB function class extension (UFX) in response to an event on the bus, such as a change in the port type or a receipt of a non-standard setup packet.

## [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_IN](#)

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the IN direction.

## [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_OUT](#)

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the OUT direction.

## [IOCTL\\_INTERNAL\\_USBFN\\_DEACTIVATE\\_USB\\_BUS](#)

Do not use.

## [IOCTL\\_INTERNAL\\_USBFN\\_DESCRIPTOR\\_UPDATE](#)

The USB function class extension sends this request to the client driver to update to the endpoint descriptor for the specified endpoint.

## [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_CLASS\\_INFO](#)

The class driver sends this request IO control code to retrieve information about the available pipes for a device, as configured in the registry.

#### [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_INTERFACE\\_DESCRIPTOR\\_SET](#)

The class driver sends this request to get the entire USB interface descriptor set for a function on the device.

#### [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_PIPE\\_STATE](#)

The class driver sends this request to get the stall state of the specified pipe.

#### [IOCTL\\_INTERNAL\\_USBFN\\_REGISTER\\_USB\\_STRING](#)

The class driver sends this request to register a USB string descriptor.

#### [IOCTL\\_INTERNAL\\_USBFN\\_RESERVED](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_RESERVED) article.

#### [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_PIPE\\_STATE](#)

The class driver sends this request to set the stall state of the specified USB pipe.

#### [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_EXIT\\_LPM](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_EXIT\_LPM) article.

#### [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_STATE](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_STATE) article.

#### [IOCTL\\_INTERNAL\\_USBFN\\_SIGNAL\\_REMOTE\\_WAKEUP](#)

The class driver sends this request to get remote wake-up notifications from endpoints.

#### [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN](#)

The class driver sends this request to initiate a data transfer to the host on the specified pipe.

#### [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN\\_APPEND\\_ZERO\\_PKT](#)

The class driver sends this request to initiate an IN transfer to the specified pipe and appends a zero-length packet to indicate the end of the transfer.

#### [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_OUT](#)

The class driver sends this request to initiate a data transfer from the host on the specified pipe.

## [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.

## [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\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED](#)

Notifies the client driver that the display out status of the DisplayPort connection has changed 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.

## [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\\_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\\_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\\_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\\_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\\_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\\_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\\_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\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#)

Learn more about: [IOCTL\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#) IOCTL

#### [IOCTL\\_UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK](#)

Sends a UCSI data block to the client driver.

#### [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_OFF](#)

The [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_OFF](#) I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_ON](#)

The [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_ON](#) I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_OFF](#)

The [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_OFF](#) I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_ON](#)

The [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_ON](#) I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#)

The [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request retrieves one

or more descriptors for the device that is associated with the indicated port index.`IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION` is a user-mode I/O control request.

#### [IOCTL\\_USB\\_GET\\_DEVICE\\_CHARACTERISTICS](#)

The client driver sends this request to determine general characteristics about a USB device, such as maximum send and receive delays for any request.

#### [IOCTL\\_USB\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#)

The `IOCTL_USB_GET_FRAME_NUMBER_AND_QPC_FOR_TIME_SYNC` IOCTL function gets the system query performance counter (QPC) value for a specific frame and microframe.

#### [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES](#)

The `IOCTL_USB_GET_HUB_CAPABILITIES` I/O control request retrieves the capabilities of a USB hub.

#### [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES\\_EX](#)

The `IOCTL_USB_GET_HUB_CAPABILITIES_EX` I/O control request retrieves the capabilities of a USB hub.`IOCTL_USB_GET_HUB_CAPABILITIES_EX` is a user-mode I/O control request. This request targets the USB hub device (`GUID_DEVINTERFACE_USB_HUB`).

#### [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#)

The `IOCTL_USB_GET_HUB_INFORMATION_EX` I/O control request is sent by an application to retrieve information about a USB hub in a `USB_HUB_INFORMATION_EX` structure. The request retrieves the highest port number on the hub.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

The `IOCTL_USB_GET_NODE_CONNECTION_ATTRIBUTES` I/O control request retrieves the Microsoft-extended port attributes for a specific port.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)

The `IOCTL_USB_GET_NODE_CONNECTION_DRIVERKEY_NAME` I/O control request retrieves the driver registry key name that is associated with the device that is connected to the indicated port.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#)

The `IOCTL_USB_GET_NODE_CONNECTION_INFORMATION` request retrieves information about the indicated USB port and the device that is attached to the port, if there is one. Client drivers must send this IOCTL at an IRQL of

`PASSIVE_LEVEL`.`IOCTL_USB_GET_NODE_CONNECTION_INFORMATION` is a user-mode I/O control request. This request targets the USB hub device (`GUID_DEVINTERFACE_USB_HUB`). Do not send this request to the root hub.

## [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request retrieves information about a USB port and the device that is attached to the port, if there is one. Client drivers must send this IOCTL at an IRQL of PASSIVE\_LEVEL. IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB). Do not send this request to the root hub.

## [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 I/O control is sent by an application to retrieve information about the protocols that are supported by a particular USB port on a hub. The request also retrieves the speed capability of the port.

## [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_NAME](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME I/O control request is used with the USB\_NODE\_CONNECTION\_NAME structure to retrieve the symbolic link name of the hub that is attached to the downstream port. IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_SUPERSPEEDPLUS\\_INFORMATION](#)

## [IOCTL\\_USB\\_GET\\_NODE\\_INFORMATION](#)

The IOCTL\_USB\_GET\_NODE\_INFORMATION I/O control request is used with the USB\_NODE\_INFORMATION structure to retrieve information about a parent device. IOCTL\_USB\_GET\_NODE\_INFORMATION is a user-mode I/O control request.

## [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

The IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request is sent by an application to retrieve information about a specific port on a USB hub.

## [IOCTL\\_USB\\_GET\\_ROOT\\_HUB\\_NAME](#)

The IOCTL\_USB\_GET\_ROOT\_HUB\_NAME I/O control request is used with the USB\_ROOT\_HUB\_NAME structure to retrieve the symbolic link name of the root hub. IOCTL\_USB\_GET\_ROOT\_HUB\_NAME is a user-mode I/O control request.

## [IOCTL\\_USB\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

The client driver sends this request to retrieve the transport characteristics.

## [IOCTL\\_USB\\_HCD\\_DISABLE\\_PORT](#)

The IOCTL\_USB\_HCD\_DISABLE\_PORT IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_ENABLE\\_PORT](#)

The IOCTL\_USB\_HCD\_ENABLE\_PORT IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_GET\\_STATS\\_1](#)

The IOCTL\_USB\_HCD\_GET\_STATS\_1 IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_GET\\_STATS\\_2](#)

The IOCTL\_USB\_HCD\_GET\_STATS\_2 IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HUB\\_CYCLE\\_PORT](#)

The IOCTL\_USB\_HUB\_CYCLE\_PORT I/O control request power-cycles the port that is associated with the PDO that receives the request.

## [IOCTL\\_USB\\_NOTIFY\\_ON\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request notifies the caller of change in transport characteristics.

## [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request registers for notifications about the changes in transport characteristics.

## [IOCTL\\_USB\\_RESET\\_HUB](#)

The IOCTL\_USB\_RESET\_HUB IOCTL is used by the USB driver stack. Do not use.

## [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

This request registers the caller with USB driver stack for time sync services.

## [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

This request unregisters the caller with USB driver stack for time sync services.

## [IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request unregisters the caller from getting notifications about transport characteristics changes.

# Enumerations

[\[+\] Expand table](#)

<a href="#">CONTROLLER_TYPE</a>	This enumeration specifies if the USB host controller is an eXtensible Host Controller Interface (xHCI) controller.
<a href="#">ENDPOINT_RESET_FLAGS</a>	Defines parameters for a request to reset an endpoint.
<a href="#">TRISTATE</a>	The TRISTATE enumeration indicates generic state values for true or false.
<a href="#">UCM_CHARGING_STATE</a>	Defines the charging state of a Type-C connector.
<a href="#">UCM_PD_CONN_STATE</a>	Defines power delivery (PD) negotiation states of a Type-C port.
<a href="#">UCM_PD_POWER_DATA_OBJECT_TYPE</a>	Defines Power Data Object types.
<a href="#">UCM_POWER_ROLE</a>	Defines power roles of USB Type-C connected devices.
<a href="#">UCM_TYPEC_CURRENT</a>	Defines different Type-C current levels, as defined in the Type-C specification.
<a href="#">UCM_TYPEC_OPERATING_MODE</a>	Defines operating modes of a USB Type-C connector.
<a href="#">UCM_TYPEC_PARTNER</a>	Defines the state of the Type-C connector.
<a href="#">UCMTCPCI_PORT_CONTROLLER_ALERT_TYPE</a>	Defines generic alert values that are used to indicate the type of hardware alert received on the port controller.
<a href="#">UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS</a>	Defines values to determine whether a display out status for a DisplayPort device is enabled.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS](#)

Defines values to determine whether a DisplayPort device is plugged in.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#)

Learn more about: [\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#)  
enumeration

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_IOCTL](#)

Defines the various device I/O control requests that are sent to the client driver for the port controller. This indicates the type of IOCTL in WPP.

## [UCMUCSI\\_PPM\\_IOCTL](#)

Defines I/O control codes handled by the client driver.

## [UCMUCSIFUNCENUM](#)

Defines values for all export functions called by a client driver of a UcmUcsiCx class extension.

## [UCSI\\_BATTERY\\_CHARGING\\_STATUS](#)

See Table 4-42, Offset 64.

## [UCSI\\_COMMAND](#)

See Table 4-51, Command Code.

## [UCSI\\_CONNECTOR\\_PARTNER\\_FLAGS](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 21.

## [UCSI\\_CONNECTOR\\_PARTNER\\_TYPE](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 29.

## [UCSI\\_GET\\_ALTERNATE\\_MODES\\_RECIPIENT](#)

Used in the GET\_ALTERNATE\_MODES command. See Table 4-24, Offset 16.

## [UCSI\\_GET\\_PDOS\\_SOURCE\\_CAPABILITIES\\_TYPE](#)

Used in the GET\_PDOS command. See Table 4-34, Offset 35.

## [UCSI\\_GET\\_PDOS\\_TYPE](#)

Used in the GET\_PDOS command. See Table 4-34, Offset 34.

<a href="#">UCSI_POWER_DIRECTION</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 20.
<a href="#">UCSI_POWER_DIRECTION_MODE</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 20.
<a href="#">UCSI_POWER_DIRECTION_ROLE</a>	Used in the SET_PDR command. The SET_PDR command is used to set the power direction dictated by the OS Policy Manager (OPM), for the current connection.
<a href="#">UCSI_POWER_OPERATION_MODE</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 16.
<a href="#">UCSI_USB_OPERATION_MODE</a>	Used in the SET_UOR command. See Table 4-18, Offset 23.
<a href="#">UCSI_USB_OPERATION_ROLE</a>	Used in the SET_UOR command. The SET_UOR command is used to set the USB operation role dictated by the OS Policy Manager (OPM), for the current connection.
<a href="#">UCX_CONTROLLER_ENDPOINT_CHARACTERISTIC_PRIORITY</a>	Indicates the priority of endpoints.
<a href="#">UCX_CONTROLLER_PARENT_BUS_TYPE</a>	The UCX_CONTROLLER_PARENT_BUS_TYPE enumeration defines the parent bus type.
<a href="#">UCX_CONTROLLER_STATE</a>	This enumeration provides values to specify the UCX controller state after a reset.
<a href="#">UCX_ENDPOINT_CHARACTERISTIC_TYPE</a>	Defines values that indicates the type of endpoint characteristic.
<a href="#">UCX_USBDEVICE_CHARACTERISTIC_TYPE</a>	Defines values that indicates the type of device characteristic.
<a href="#">UCX_USBDEVICE_RECOVERY_ACTION</a>	Defines values for FLDR and PLDR trigger resets.

## [UDECX\\_ENDPOINT\\_TYPE](#)

Defines values for endpoint types supported by a virtual USB device.

## [UDECX\\_ENDPOINTS\\_CONFIGURE\\_TYPE](#)

Defines values for endpoint configuration options.

## [UDECX\\_USB\\_DEVICE\\_FUNCTION\\_POWER](#)

Defines values for function wake capability of a virtual USB 3.0 device.

## [UDECX\\_USB\\_DEVICE\\_SPEED](#)

Defines values for USB device speeds.

## [UDECX\\_USB\\_DEVICE\\_WAKE\\_SETTING](#)

Defines values for remote wake capability of a virtual USB device.

## [UDECX\\_WDF\\_DEVICE\\_RESET\\_ACTION](#)

Defines values that indicate the types of reset operation supported by an emulated USB host controller.

## [UDECX\\_WDF\\_DEVICE\\_RESET\\_TYPE](#)

Defines values that indicates the type of reset for a UDE device.

## [UFX\\_CLASS\\_FUNCTIONS](#)

Learn more about: [\\_UFX\\_CLASS\\_FUNCTIONS](#) enumeration

## [URS\\_HARDWARE\\_EVENT](#)

Defines values for the hardware events that a client driver for a USB dual-role controller can report.

## [URS\\_HOST\\_INTERFACE\\_TYPE](#)

Defines values for the various types of USB host controllers.

## [URS\\_ROLE](#)

Defines values for roles supported by a USB dual-role controller.

## [USB\\_CONNECTION\\_STATUS](#)

The USB\_CONNECTION\_STATUS enumerator indicates the status of the connection to a device on a USB hub port.

#### [USB\\_CONTROLLER\\_FLAVOR](#)

The USB\_CONTROLLER\_FLAVOR enumeration specifies the type of USB host controller.

#### [USB\\_DEVICE\\_SPEED](#)

The USB\_DEVICE\_SPEED enumeration defines constants for USB device speeds.

#### [USB\\_HUB\\_NODE](#)

The USB\_HUB\_NODE enumerator indicates whether a device is a hub or a composite device.

#### [USB\\_HUB\\_TYPE](#)

The USB\_HUB\_TYPE enumeration defines constants that indicate the type of USB hub. The hub type is retrieved by the IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX I/O control request.

#### [USB\\_NOTIFICATION\\_TYPE](#)

Learn more about: [\\_USB\\_NOTIFICATION\\_TYPE](#) enumeration

#### [USBC\\_CHARGING\\_STATE](#)

Learn how USBC\_CHARGING\_STATE defines the charging state of a Type-C connector.

#### [USBC\\_CURRENT](#)

Learn how USBC\_CURRENT defines different Type-C current levels, as defined in the Type-C specification.

#### [USBC\\_DATA\\_ROLE](#)

Defines data roles of USB Type-C connected devices.

#### [USBC\\_PARTNER](#)

Defines values for the type of connector partner detected on the USB Type-C connector.

#### [USBC\\_PD\\_AUGMENTED\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

Learn how USBC\_PD\_AUGMENTED\_POWER\_DATA\_OBJECT\_TYPE defines augmented power data object (APDO) types.

#### [USBC\\_PD\\_CONN\\_STATE](#)

Learn how USBC\_PD\_CONN\_STATE defines power delivery (PD) negotiation states of a Type-C port.

#### [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

Learn how USBC\_PD\_POWER\_DATA\_OBJECT\_TYPE defines power data object (PDO) types.

#### [USBC\\_POWER\\_ROLE](#)

Learn how USBC\_POWER\_ROLE defines power roles of USB Type-C connected devices.

#### [USBC\\_TYPEC\\_OPERATING\\_MODE](#)

Learn how USBC\_TYPEC\_OPERATING\_MODE defines operating modes of a USB Type-C connector.

#### [USBC\\_UCSI\\_SET\\_POWER\\_LEVEL\\_C\\_CURRENT](#)

Defines values for current power operation mode.

#### [USBD\\_ENDPOINT\\_OFFLOAD\\_MODE](#)

Defines values for endpoint offloading options in the USB device or host controller.

#### [USBD\\_PIPE\\_TYPE](#)

The USBD\_PIPE\_TYPE enumerator indicates the type of pipe.

#### [USBFN\\_ACTION](#)

Defines special actions UFX should take when the client driver calls the UfxDevicePortDetectCompleteEx function.

#### [USBFN\\_ATTACH\\_ACTION](#)

Defines the actions that the Universal Serial Bus (USB) function stack takes when a device is attached to a USB port.

#### [USBFN\\_BUS\\_SPEED](#)

The USBFN\_BUS\_SPEED enumeration defines possible bus speeds.

#### [USBFN\\_DEVICE\\_STATE](#)

Defines the Universal Serial Bus (USB) device states for the device/controller. These states correspond to the USB device states as defined in section 9.1 of the USB 2.0 Specification.

#### [USBFN\\_DIRECTION](#)

Defines the USB data transfer direction types.

<a href="#">USBFN_EVENT</a>
Defines notifications sent to class drivers.
<a href="#">USBFN_PORT_TYPE</a>
Defines the possible port types that can be returned by the client driver during port detection.
<a href="#">USBPM_ACCESS_TYPE</a>
Defines the access types for calling Policy Manager functions.
<a href="#">USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT</a>
Defines format values used in <code>USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS</code> .
<a href="#">USBPM_EVENT_TYPE</a>
Defines values for types of events.

## Functions

[+] [Expand table](#)

<a href="#">COMPOSITE_DEVICE_CAPABILITIES_INIT</a>
The <code>COMPOSITE_DEVICE_CAPABILITIES_INIT</code> macro initializes the <code>COMPOSITE_DEVICE_CAPABILITIES</code> structure.
<a href="#">EVT_UCM_CONNECTOR_SET_DATA_ROLE</a>
The client driver's implementation of the <code>EVT_UCM_CONNECTOR_SET_DATA_ROLE</code> event callback function that swaps the data role of the connector to the specified role when attached to a partner connector.
<a href="#">EVT_UCM_CONNECTOR_SET_POWER_ROLE</a>
The client driver's implementation of the <code>EVT_UCM_CONNECTOR_SET_POWER_ROLE</code> event callback function that sets the power role of the connector to the specified role when attached to a partner connector.
<a href="#">EVT_UCX_CONTROLLER_GET_CURRENT_FRAMENUMBER</a>
The client driver's implementation that UCX calls to retrieve the current 32-bit frame number.

## [EVT\\_UCX\\_CONTROLLER\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback to retrieves the system query performance counter (QPC) value synchronized with the frame and microframe.

## [EVT\\_UCX\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

UCX invokes this callback to retrieve the host controller characteristics.

## [EVT\\_UCX\\_CONTROLLER\\_QUERY\\_USB\\_CAPABILITY](#)

The client driver's implementation to determine if the controller supports a specific capability.

## [EVT\\_UCX\\_CONTROLLER\\_RESET](#)

The client driver's implementation that UCX calls to reset the controller.

## [EVT\\_UCX\\_CONTROLLER\\_SET\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#)

UCX invokes this callback function to specify its preference in transport characteristics for which the client driver must send notifications when changes occur.

## [EVT\\_UCX\\_CONTROLLER\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback function to the start time tracking functionality in the controller.

## [EVT\\_UCX\\_CONTROLLER\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback function to the stop time tracking functionality in the controller.

## [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#)

The client driver's implementation that UCX calls when a new USB device is detected.

## [EVT\\_UCX\\_DEFAULT\\_ENDPOINT\\_UPDATE](#)

The client driver's implementation that UCX calls with information about the default endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_ABORT](#)

The client driver's implementation that UCX calls to abort the queue associated with the endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_GET\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#)

UCX invokes this callback function to get information about transfer path delays for an isochronous endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_OK\\_TO\\_CANCEL\\_TRANSFERS](#)

The client driver's implementation that UCX calls to notify the controller driver that it can complete cancelled transfers on the endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_PURGE](#)

The client driver's implementation that completes all outstanding I/O requests on the endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_RESET](#)

The client driver's implementation that UCX calls to reset the controller's programming for an endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_SET\\_CHARACTERISTIC](#)

UCX invokes this callback function to set the priority on an endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_START](#)

The client driver's implementation that UCX calls to start the queue associated with the endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ADD](#)

The client driver's implementation that UCX calls to create static streams.

#### [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_DISABLE](#)

The client driver's implementation that UCX calls to release controller resources for all streams for an endpoint.

#### [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ENABLE](#)

The client driver's implementation that UCX calls to enable the static streams.

#### [EVT\\_UCX\\_ROOTHUB\\_CONTROL\\_URB](#)

The client driver uses this callback type to implement handlers that UCX calls when it receives feature control requests on the USB hub.

#### [EVT\\_UCX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about USB 2.0 ports on the root hub.

#### [EVT\\_UCX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about USB 3.0 ports on the root hub.

#### [EVT\\_UCX\\_ROOTHUB\\_GET\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about the root hub.

#### [EVT\\_UCX\\_ROOTHUB\\_INTERRUPT\\_TX](#)

The client driver's implementation that UCX calls when it receives a request for information about changed ports.

#### [EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#)

The client driver's implementation that UCX calls to address the USB device.

#### [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

The client driver's implementation that UCX calls to add a new default endpoint for a USB device.

#### [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#)

The client driver's implementation that UCX calls to release controller resources associated with the device and its default endpoint.

#### [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#)

The client driver's implementation that UCX calls to program information about the device and its default control endpoint into the controller.

#### [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#)

The client driver's implementation that UCX calls to add a new endpoint for a USB device.

#### [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#)

The client driver's implementation that UCX calls to configure endpoints in the controller.

#### [EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#)

UCX invokes this callback to retrieve the device characteristics.

#### [EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#)

The client driver's implementation that UCX calls to retrieve hub properties.

#### [EVT\\_UCX\\_USBDEVICE\\_RESET](#)

The client driver's implementation that UCX calls when the port to which the device is attached is reset.

#### [EVT\\_UCX\\_USBDEVICE\\_RESUME](#)

UCX invokes this callback function to resume a device from suspend state.

#### [EVT\\_UCX\\_USBDEVICE\\_SUSPEND](#)

UCX invokes this callback function to send a device suspend state.

#### [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#)

The client driver's implementation that UCX calls to update device properties.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to bring the virtual USB device out of a low power state to working state.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to send the virtual USB device to a low power state.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create the default control endpoint on the virtual USB device.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINT\\_ADD](#)

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create a dynamic endpoint on the virtual USB device.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)

The USB device emulation class extension (UdeCx) invokes this callback function to change the configuration by selecting an alternate setting, disabling current endpoints, or adding dynamic endpoints.

#### [EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to change the function state of the specified interface of the virtual USB 3.0 device.

#### [EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#)

The USB device emulation class extension (UdeCx) invokes this callback function to stop queuing I/O requests to the endpoint's queue and cancel unprocessed requests.

#### [EVT\\_UDECX\\_USB\\_ENDPOINT\\_RESET](#)

The USB device emulation class extension (UdeCx) invokes this callback function to reset an endpoint of the virtual USB device.

#### [EVT\\_UDECX\\_USB\\_ENDPOINT\\_START](#)

The USB device emulation class extension (UdeCx) invokes this callback function to start processing I/O requests on the specified endpoint of the virtual USB device.

#### [EVT\\_UDECX\\_WDF\\_DEVICE\\_QUERY\\_USB\\_CAPABILITY](#)

The UDE client driver's implementation to determine the capabilities that are supported by the emulated USB host controller.

#### [EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#)

The UDE client driver's implementation to reset the emulated host controller or the devices attached to it.

#### [EVT\\_UFX\\_DEVICE\\_ADDRESSED](#)

The client driver's implementation to assign an address on the function controller.

#### [EVT\\_UFX\\_DEVICE\\_CONTROLLER\\_RESET](#)

The client driver's implementation to reset the function controller to its initial state.

#### [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

The client driver's implementation to create a default control endpoint.

#### [EVT\\_UFX\\_DEVICE\\_ENDPOINT\\_ADD](#)

The client driver's implementation to create a default endpoint object.

#### [EVT\\_UFX\\_DEVICE\\_HOST\\_CONNECT](#)

The client driver's implementation to initiate connection with the host.

#### [EVT\\_UFX\\_DEVICE\\_HOST\\_DISCONNECT](#)

The client driver's implementation to disable the function controller's communication with the host.

#### [EVT\\_UFX\\_DEVICE\\_PORT\\_CHANGE](#)

The client driver's implementation to update the type of the new port to which the USB device is connected.

#### [EVT\\_UFX\\_DEVICE\\_PORT\\_DETECT](#)

	The client driver's implementation to initiate port detection.
<a href="#">EVT_UFX_DEVICE_PROPRIETARY_CHARGER_DETECT</a>	The client driver's implementation to initiate proprietary charger detection.
<a href="#">EVT_UFX_DEVICE_PROPRIETARY_CHARGER_RESET</a>	The client driver's implementation to resets proprietary charger.
<a href="#">EVT_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY</a>	The client driver's implementation to set charger information that it uses to enable charging over USB.
<a href="#">EVT_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL</a>	The client driver's implementation to initiate remote wake-up on the function controller.
<a href="#">EVT_UFX_DEVICE_SUPER_SPEED_POWER_FEATURE</a>	The client driver's implementation to set or clear the specified power feature on the function controller.
<a href="#">EVT_UFX_DEVICE_TEST_MODE_SET</a>	The client driver's implementation to set the test mode of the function controller.
<a href="#">EVT_UFX_DEVICE_TESTHOOK</a>	This IOCTL code is not supported.
<a href="#">EVT_UFX_DEVICE_USB_STATE_CHANGE</a>	The client driver's implementation to update the state of the USB device.
<a href="#">EVT_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS</a>	The USB dual-role class extension invokes this callback to allow the client driver to insert the resources from the resource-requirements-list object to resource lists that will be used during the life time of each role.
<a href="#">EVT_URS_SET_ROLE</a>	The URS class extension invokes this event callback when it requires the client driver to change the role of the controller.
<a href="#">EVT_USBPM_EVENT_CALLBACK</a>	

Sends notifications about hub arrival/removal and connector state changes.

#### [GET\\_ISO\\_URB\\_SIZE](#)

The GET\_ISO\_URB\_SIZE macro returns the number of bytes required to hold an isochronous transfer request.

#### [PUSB\\_BUSIFFN\\_ENUM\\_LOG\\_ENTRY](#)

This callback function is not supported. The EnumLogEntry routine makes a log entry.

#### [PUSB\\_BUSIFFN\\_GETUSBDI\\_VERSION](#)

The GetUSBdiVersion routine returns the USB interface version number and the version number of the USB specification that defines the interface, along with information about host controller capabilities.

#### [PUSB\\_BUSIFFN\\_IS\\_DEVICE\\_HIGH\\_SPEED](#)

The USB\_Busiffn\_Is\_Device\_High\_Speed routine returns TRUE if the device is operating at high speed.

#### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_INFORMATION](#)

The QueryBusInformation routine gets information about the bus.

#### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_TIME](#)

The QueryBusTime function gets the current 32-bit USB frame number.

#### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_TIME\\_EX](#)

The QueryBusTimeEx routine gets the current 32-bit USB micro-frame number.

#### [PUSB\\_BUSIFFN\\_QUERY\\_CONTROLLER\\_TYPE](#)

The QueryControllerType routine gets information about the USB host controller to which the USB device is attached.

#### [PUSB\\_BUSIFFN\\_SUBMIT\\_ISO\\_OUT\\_URB](#)

This callback function is not supported. The SubmitIsoOutUrb function submits a USB request block (URB) directly to the bus driver without requiring the allocation of an IRP.

#### [UCM\\_CONNECTOR\\_CONFIG\\_INIT](#)

Initializes a UCM\_CONNECTOR\_CONFIG structure.

#### [UCM\\_CONNECTOR\\_PD\\_CONFIG\\_INIT](#)

	Initializes a UCM_CONNECTOR_PD_CONFIG structure.
<a href="#">UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS_INIT</a>	Initializes a UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS structure.
<a href="#">UCM_CONNECTOR_TYPEC_ATTACH_PARAMS_INIT</a>	Initializes a UCM_CONNECTOR_TYPEC_ATTACH_PARAMS structure.
<a href="#">UCM_CONNECTOR_TYPEC_CONFIG_INIT</a>	Initializes the UCM_CONNECTOR_TYPEC_CONFIG structure.
<a href="#">UCM_MANAGER_CONFIG_INIT</a>	Initializes a UCM_MANAGER_CONFIG structure.
<a href="#">UCM_PD_POWER_DATA_OBJECT_GET_TYPE</a>	Retrieves the type of Power Data Object from the UCM_PD_POWER_DATA_OBJECT structure.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_BATTERY</a>	Initializes a UCM_PD_POWER_DATA_OBJECT structure as a Battery Supply type Power Data Object.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_FIXED</a>	Initializes a to the UCM_PD_POWER_DATA_OBJECT for a Fixed Supply type Power Data Object.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT ULONG</a>	Initializes a UCM_PD_POWER_DATA_OBJECT structure by interpreting Power Data Object values and sets each field correctly.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_VARIABLE_NON_BATTERY</a>	Initializes a UCM_PD_POWER_DATA_OBJECT structure as a Variable Supply Non Battery type Power Data Object.
<a href="#">UCM_PD_REQUEST_DATA_OBJECT_INIT ULONG</a>	Initializes a UCM_PD_REQUEST_DATA_OBJECT structure by interpreting Request Data Object values and sets each field correctly.
<a href="#">UcmConnectorChargingStateChanged</a>	Notifies the USB connector manager framework extension (UcmCx) with the updated charging state of the partner connector.

<a href="#">UcmConnectorCreate</a>
Creates a connector object.
<a href="#">UcmConnectorDataDirectionChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the new data role of a change in data role.
<a href="#">UcmConnectorPdConnectionStateChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the connection capabilities of the currently negotiated PD contract (if any).
<a href="#">UcmConnectorPdPartnerSourceCaps</a>
Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the partner connector.
<a href="#">UcmConnectorPdSourceCaps</a>
Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the connector.
<a href="#">UcmConnectorPowerDirectionChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the new power role of the partner connector.
<a href="#">UcmConnectorTypeCAttach</a>
Notifies the USB connector manager framework extension (UcmCx) when a partner connector is attached.
<a href="#">UcmConnectorTypeCCurrentAdChanged</a>
Notifies the USB connector manager framework extension (UcmCx) when the specified connector changes the current advertisement. Either the connector changes it (when it is DFP/Source), or the partner changed it (when it is UFP/Sink).
<a href="#">UcmConnectorTypeCDetach</a>
Notifies the USB connector manager framework extension (UcmCx) when the partner connector detaches from the specified Type-C connector.
<a href="#">UcmInitializeDevice</a>
Initializes the USB connector manager framework extension (UcmCx).

<a href="#">UCMTCPCI_DEVICE_CONFIG_INIT</a>	Initializes the UCMTCPCI_DEVICE_CONFIG structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_ALERT_DATA_INIT</a>	Initializes the UCMTCPCI_PORT_CONTROLLER_ALERT_DATA structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_CAPABILITIES_INIT</a>	Initializes the UCMTCPCI_PORT_CONTROLLER_CAPABILITIES structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_CONFIG_INIT</a>	Initializes the UCMTCPCI_PORT_CONTROLLER_CONFIG structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION_INIT</a>	Initializes the UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION structure.
<a href="#">UcmTpcDiDeviceInitialize</a>	Initializes the USB Type-C Port Controller Interface framework extension (UcmTpcCx).
<a href="#">UcmTpcDiDeviceInitInitialize</a>	Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UcmTpcPortControllerAlert</a>	Sends information about the hardware alerts that are received on the port controller to UcmTpcCx.
<a href="#">UcmTpcPortControllerCreate</a>	Creates a port controller object to register with UcmTpcCx.
<a href="#">UcmTpcPortControllerSetHardwareRequestQueue</a>	Assigns a framework queue object to which the UcmTpcCx dispatches hardware requests for the port controller.
<a href="#">UcmTpcPortControllerStart</a>	Indicates to the UcmTpcCx class extension that the client driver is now ready to service hardware requests for the port controller.

## [UcmTcpciPortControllerStop](#)

Indicates to the UcmTcpciCx class extension to stop sending hardware requests to the port controller object.

## [UCMUCSI\\_CONNECTOR\\_INFO\\_INIT](#)

Initializes a UCMUCSI\_CONNECTOR\_INFO structure.

## [UCMUCSI\\_DEVICE\\_CONFIG\\_INIT](#)

Initializes a UCMUCSI\_DEVICE\_CONFIG structure.

## [UCMUCSI\\_PPM\\_CONFIG\\_INIT](#)

Initializes a UCMUCSI\_PPM\_CONFIG structure.

## [UcmUcsiConnectorCollectionAddConnector](#)

Adds a connector to the connector collection object.

## [UcmUcsiConnectorCollectionCreate](#)

Creates a connector collection object with UcmUcsiCx.

## [UcmUcsiDeviceInitialize](#)

Initializes the UCSI extension (UcmUcsiCx).

## [UcmUcsiDeviceInitInitialize](#)

Initializes the WDFDEVICE\_INIT provided by the framework.

## [UcmUcsiPpmCreate](#)

Creates a Platform Policy Manager (PPM) object.

## [UcmUcsiPpmNotification](#)

Informs the UcmUcsiCx class extension about a UCSI notification.

## [UcmUcsiPpmSetUcsiCommandRequestQueue](#)

Provides a framework queue object that is used to dispatch UCSI commands to the client driver.

## [UcmUcsiPpmStart](#)

Instructs the class extension to start sending requests to the client driver.

## [UcmUcsiPpmStop](#)

Instructs the class extension to stop sending requests to the client driver.

## [UCSI\\_CMD\\_SUCCEEDED](#)

On successful completion of a UCSI command the PPM firmware fills the CCI Data Structure provided by the client driver.

## [UCX\\_CONTROLLER\\_CONFIG\\_SET\\_ACPI\\_INFO](#)

Initializes a UCX\_CONTROLLER\_CONFIG structure with the specified values for the controller with ACPI as the parent.

## [UCX\\_CONTROLLER\\_CONFIG\\_SET\\_PCI\\_INFO](#)

Initializes a UCX\_CONTROLLER\_CONFIG structure with the specified values for the controller with PCI as the parent bus type.

## [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

Initializes a UCX\_DEFAULT\_ENDPOINT\_EVENT\_CALLBACKS structure with client driver's callback functions. The client driver calls this function before calling UcxEndpointCreate method to create an endpoint and register its callback functions with UCX.

## [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

Initializes a UCX\_ENDPOINT\_EVENT\_CALLBACKS structure with client driver's callback functions. The client driver calls this function before calling UcxEndpointCreate method to create an endpoint and register its callback functions with UCX.

## [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#)

Initializes a UCX\_USBDEVICE\_EVENT\_CALLBACKS structure with the function pointers to client driver's callback functions.

## [UcxControllerCreate](#)

Creates a host controller object.

## [UcxControllerNeedsReset](#)

Initiates a non-Plug and Play (PnP) controller reset operation by queuing an event into the controller reset state machine.

## [UcxControllerNotifyTransportCharacteristicsChange](#)

Notifies UCX about a new port change event from host controller.

<a href="#">UcxControllerResetComplete</a>	Informs USB host controller extension (UCX) that the reset operation has completed.
<a href="#">UcxControllerSetFailed</a>	Informs USB Host Controller Extension (UCX) that the controller has encountered a critical failure.
<a href="#">UcxControllerSetIdStrings</a>	Updates the identifier strings of a controller after the controller has been initialized.
<a href="#">UcxDefaultEndpointInitSetEventCallbacks</a>	Initializes a UCXENDPOINT_INIT structure with client driver's event callback functions related to the default endpoint.
<a href="#">UcxEndpointAbortComplete</a>	Notifies UCX that a transfer abort operation has been completed on the specified endpoint object.
<a href="#">UcxEndpointCreate</a>	Creates an endpoint on the specified USB device object.
<a href="#">UcxEndpointGetStaticStreamsReferenced</a>	Returns a referenced static streams object for the specified endpoint.
<a href="#">UcxEndpointInitSetEventCallbacks</a>	Initializes a UCXENDPOINT_INIT structure with client driver's event callback functions related to endpoints on the device.
<a href="#">UcxEndpointNeedToCancelTransfers</a>	The client driver calls this method before it cancels transfers on the wire.
<a href="#">UcxEndpointNoPingResponseError</a>	Notifies UCX about a "No Ping Response" error for a transfer on the specified endpoint object.
<a href="#">UcxEndpointPurgeComplete</a>	Notifies UCX that a purge operation has been completed on the specified endpoint object.
<a href="#">UcxEndpointSetWdfIoQueue</a>	Sets a framework queue on the specified endpoint object.

<a href="#">UcxInitializeDeviceInit</a>	UcxInitializeDeviceInit initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UcxIoDeviceControl</a>	Allows USB host controller extension (UCX) to handle an I/O control code (IOCTL) request from user mode.
<a href="#">UcxRootHubPortChanged</a>	Notifies UCX about a new port change event on the host controller.
<a href="#">UcxStaticStreamsCreate</a>	Creates a static streams object.
<a href="#">UcxStaticStreamsSetStreamInfo</a>	Sets stream information for each stream enabled by the client driver.
<a href="#">UcxUsbDeviceCreate</a>	Creates a USB device object on the specified controller.
<a href="#">UcxUsbDeviceInitSetEventCallbacks</a>	Initializes a UCXUSBDEVICE_INIT structure with client driver's event callback functions.
<a href="#">UcxUsbDeviceRemoteWakeNotification</a>	Notifies UCX that a remote wake signal from the device is received.
<a href="#">UDECX_USB_DEVICE_CALLBACKS_INIT</a>	Initializes a UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS structure before a UdecxUsbDeviceCreate call.
<a href="#">UDECX_USB_DEVICE_PLUG_IN_OPTIONS_INIT</a>	Initializes a UDECX_USB_DEVICE_PLUG_IN_OPTIONS structure.
<a href="#">UDECX_USB_ENDPOINT_CALLBACKS_INIT</a>	Initializes a UDECX_USB_ENDPOINT_CALLBACKS structure before a UdecxUsbEndpointCreate call.
<a href="#">UDECX_WDF_DEVICE_CONFIG_INIT</a>	

	Initializes a UDECX_WDF_DEVICE_CONFIG structure.
<a href="#">UdecxInitializeWdfDeviceInit</a>	UdecxInitializeWdfDeviceInit initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UdecxUrbComplete</a>	Completes the URB request with a USB-specific completion status code.
<a href="#">UdecxUrbCompleteWithNtStatus</a>	Completes the URB request with an NTSTATUS code.
<a href="#">UdecxUrbRetrieveBuffer</a>	Retrieves the transfer buffer of an URB from the specified framework request object sent to the endpoint queue.
<a href="#">UdecxUrbRetrieveControlSetupPacket</a>	Retrieves a USB control setup packet from a specified framework request object.
<a href="#">UdecxUrbSetBytesCompleted</a>	Sets the number of bytes transferred for the URB contained within a framework request object.
<a href="#">UdecxUsbDeviceCreate</a>	Creates a USB Device Emulation (UDE) device object.
<a href="#">UdecxUsbDeviceInitAddDescriptor</a>	Adds a USB descriptor to the initialization parameters used to create a virtual USB device.
<a href="#">UdecxUsbDeviceInitAddDescriptorWithIndex</a>	Learn how the UdecxUsbDeviceInitAddDescriptorWithIndex function adds a USB descriptor to the initialization parameters used to create a virtual USB device.
<a href="#">UdecxUsbDeviceInitAddStringDescriptor</a>	Adds a USB string descriptor to the initialization parameters used to create a virtual USB device.
<a href="#">UdecxUsbDeviceInitAddStringDescriptorRaw</a>	Learn how this method adds a USB string descriptor to the initialization parameters used to create a virtual USB device.

<a href="#">UdecxUsbDeviceInitAllocate</a>	Allocates memory for a UDECXUSBDEVICE_INIT structure that is used to initialize a virtual USB device.
<a href="#">UdecxUsbDeviceInitFree</a>	Releases the resources that were allocated by the UdecxUsbDeviceInitAllocate call.
<a href="#">UdecxUsbDeviceInitSetEndpointsType</a>	Indicates the type of endpoint (simple or dynamic) in the initialization parameters that the client driver uses to create the virtual USB device.
<a href="#">UdecxUsbDeviceInitSetSpeed</a>	Sets the USB speed of the virtual USB device to create.
<a href="#">UdecxUsbDeviceInitSetStateChangeCallbacks</a>	Initializes a WDF-allocated structure with pointers to callback functions.
<a href="#">UdecxUsbDeviceLinkPowerEntryComplete</a>	Completes an asynchronous request for bringing the device out of a low power state.
<a href="#">UdecxUsbDeviceLinkPowerExitComplete</a>	Completes an asynchronous request for sending the device to a low power state.
<a href="#">UdecxUsbDevicePlugIn</a>	Notifies the USB device emulation class extension (UdeCx) that the USB device has been plugged in the specified port.
<a href="#">UdecxUsbDevicePlugOutAndDelete</a>	Disconnects the virtual USB device.
<a href="#">UdecxUsbDeviceSetFunctionSuspendAndWakeComplete</a>	Completes an asynchronous request for changing the power state of a particular function of a virtual USB 3.0 device.
<a href="#">UdecxUsbDeviceSignalFunctionWake</a>	Initiates wake up of the specified function from a low power state. This applies to virtual USB 3.0 devices.

<a href="#">UdecxUsbDeviceSignalWake</a>	Initiates wake up from a low link power state for a virtual USB 2.0 device.
<a href="#">UdecxUsbEndpointCreate</a>	Creates a UDE endpoint object.
<a href="#">UdecxUsbEndpointInitFree</a>	Release the resources that were allocated by the UdecxUsbSimpleEndpointInitAllocate call.
<a href="#">UdecxUsbEndpointInitSetCallbacks</a>	Sets pointers to UDE client driver-implemented callback functions in the initialization parameters of the simple endpoint to create.
<a href="#">UdecxUsbEndpointInitSetEndpointAddress</a>	Sets the address of the endpoint in the initialization parameters of the simple endpoint to create.
<a href="#">UdecxUsbEndpointPurgeComplete</a>	Completes an asynchronous request for canceling all I/O requests queued to the specified endpoint.
<a href="#">UdecxUsbEndpointSetWdfIoQueue</a>	Sets a framework queue object with a UDE endpoint.
<a href="#">UdecxUsbSimpleEndpointInitAllocate</a>	Allocates memory for an initialization structure that is used to create a simple endpoint for the specified virtual USB device.
<a href="#">UdecxWdfDeviceAddUsbDeviceEmulation</a>	Initializes a framework device object to support operations related to a host controller and a virtual USB device attached to the controller.
<a href="#">UdecxWdfDeviceNeedsReset</a>	Informs the USB device emulation class extension (UdeCx) that the device needs a reset operation.
<a href="#">UdecxWdfDeviceResetComplete</a>	Informs the USB device emulation class extension (UdeCx) that the reset operation on the specified controller has completed.

<a href="#">UdecxWdfDeviceTryHandleUserIoctl</a>	Attempts to handle an IOCTL request sent by a user-mode software.
<a href="#">UFX_DEVICE_CALLBACKS_INIT</a>	The UFX_DEVICE_CALLBACKS_INIT macro initializes the UFX_DEVICE_CALLBACKS structure.
<a href="#">UFX_DEVICE_CAPABILITIES_INIT</a>	The UFX_DEVICE_CAPABILITIES_INIT macro initializes the UFX_DEVICE_CAPABILITIES structure.
<a href="#">UFX_ENDPOINT_CALLBACKS_INIT</a>	The UFX_ENDPOINT_CALLBACKS_INIT macro initializes the UFX_ENDPOINT_CALLBACKS structure.
<a href="#">UFX_PROPRIETARY_CHARGER_ABORT_OPERATION</a>	The filter driver's implementation to abort a charger operation.
<a href="#">UFX_PROPRIETARY_CHARGER_DETECT</a>	The filter driver's implementation to detect if a charger is attached and get details about the charger.
<a href="#">UFX_PROPRIETARY_CHARGER_RESET_OPERATION</a>	The filter driver's implementation to reset a charger operation.
<a href="#">UFX_PROPRIETARY_CHARGER_SET_PROPERTY</a>	The filter driver's implementation to set a configurable property on the charger.
<a href="#">UfxDeviceCreate</a>	Creates a UFX device object, registers event callback routines, and specifies capabilities specific to the controller.
<a href="#">UfxDeviceEventComplete</a>	Informs UFX that the client driver has completed processing a UFX callback function.
<a href="#">UfxDeviceIoControl</a>	Passes non-internal IOCTLs from user-mode to UFX.
<a href="#">UfxDeviceIoInternalControl</a>	Passes kernel mode IOCTLs to UFX.

<a href="#">UfxDeviceNotifyAttach</a>
Notifies UFX that the device's USB cable has been attached.
<a href="#">UfxDeviceNotifyDetach</a>
Notifies UFX that the device's USB cable has been detached.
<a href="#">UfxDeviceNotifyFinalExit</a>
Notifies UFX that the device is detached.
<a href="#">UfxDeviceNotifyHardwareFailure</a>
Notifies UFX about a non-recoverable hardware failure in the controller.
<a href="#">UfxDeviceNotifyHardwareReady</a>
Notifies UFX that the hardware is ready.
<a href="#">UfxDeviceNotifyReset</a>
Notifies UFX about a USB bus reset event.
<a href="#">UfxDeviceNotifyResume</a>
Notifies UFX about a USB bus resume event.
<a href="#">UfxDeviceNotifySuspend</a>
Notifies UFX about a USB bus suspend event.
<a href="#">UfxDevicePortDetectComplete</a>
Notifies UFX about the port type that was detected.
<a href="#">UfxDevicePortDetectCompleteEx</a>
Notifies UFX about the port type that was detected, and optionally requests an action.
<a href="#">UfxDeviceProprietaryChargerDetectComplete</a>
Notifies UFX about a detected proprietary port/charger type.
<a href="#">UfxEndpointCreate</a>
Creates an endpoint object.
<a href="#">UfxEndpointGetCommandQueue</a>

Returns the command queue previously created by UfxEndpointCreate.
<a href="#">UfxEndpointGetTransferQueue</a>
Returns the transfer queue previously created by UfxEndpointCreate.
<a href="#">UfxEndpointInitSetEventCallbacks</a>
Initialize a UFXENDPOINT_INIT structure.
<a href="#">UfxEndpointNotifySetup</a>
Notifies UFX when the client driver receives a setup packet from the host.
<a href="#">UfxFdInit</a>
Initializes the WDFDEVICE_INIT structure that the client driver subsequently provides when it calls WdfDeviceCreate.
<a href="#">URS_CONFIG_INIT</a>
Initializes a URS_CONFIG structure.
<a href="#">UrsDeviceInitialize</a>
Initializes a framework device object to support operations related to a USB dual-role controller and registers the relevant event callback functions with the USB dual-role controller class extension.
<a href="#">UrsDeviceInitInitialize</a>
Learn how this function initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UrsIoResourceListAppendDescriptor</a>
Appends the specified resource descriptor to the specified I/O resource list object that maintains resource descriptors for the host or function role.
<a href="#">UrsReportHardwareEvent</a>
Notifies the USB dual-role class extension about a new hardware event.
<a href="#">UrsSetHardwareEventSupport</a>
Indicates the client driver's support for reporting new hardware events.
<a href="#">UrsSetPoHandle</a>

Registers and deletes the client driver's registration with the power management framework (PoFx).

#### [UsbBuildGetStatusRequest](#)

The UsbBuildGetStatusRequest macro formats an URB to obtain status from a device, interface, endpoint, or other device-defined target on a USB device.

#### [UsbBuildInterruptOrBulkTransferRequest](#)

The UsbBuildInterruptOrBulkTransferRequest macro formats an URB to send or receive data on a bulk pipe, or to receive data from an interrupt pipe.

#### [UsbBuildOpenStaticStreamsRequest](#)

The UsbBuildOpenStaticStreamsRequest inline function formats an URB structure for an open-streams request. The request opens streams associated with the specified bulk endpoint.

#### [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_GET\\_TYPE](#)

Retrieves the type of Power Data Object (PDO).

#### [USBC\\_START\\_DEVICE\\_CALLBACK](#)

The USBC\_START\_DEVICE\_CALLBACK routine allows a USB client driver to provide a custom definition of the interface collections on a device.

#### [USBD\\_AssignUrbToStackLocation](#)

The USBD\_AssignUrbToStackLocation routine is called by a client driver to associate an URB with the IRP's next stack location.

#### [USBD\\_BuildRegisterCompositeDevice](#)

The USBD\_BuildRegisterCompositeDevice routine is called by the driver of a USB multi-function device (composite driver) to initialize a REGISTER\_COMPOSITE\_DEVICE structure with the information required for registering the driver with the USB driver stack.

#### [USBD\\_CalculateUsbBandwidth](#)

The USBD\_CalculateUsbBandwidth routine has been deprecated in Windows XP and later operating systems. Do not use.

#### [USBD\\_CloseHandle](#)

The USBD\_CloseHandle routine is called by a USB client driver to close a USBD handle and release all resources associated with the driver's registration.

#### [USBD\\_CreateConfigurationRequest](#)

The USBD\_CreateConfigurationRequest routine has been deprecated. Use USBD\_CreateConfigurationRequestEx instead.

#### [USBD\\_CreateConfigurationRequestEx](#)

The USBD\_CreateConfigurationRequestEx routine allocates and formats a URB to select a configuration for a USB device.USBD\_CreateConfigurationRequestEx replaces USBD\_CreateConfigurationRequest.

#### [USBD\\_CreateHandle](#)

The USBD\_CreateHandle routine is called by a WDM USB client driver to obtain a USBD handle. The routine registers the client driver with the underlying USB driver stack.

#### [USBD\\_GetInterfaceLength](#)

The USBD\_GetInterfaceLength routine obtains the length of a given interface descriptor, including the length of all endpoint descriptors contained within the interface.

#### [USBD\\_GetPdoRegistryParameter](#)

The USBD\_GetPdoRegistryParameter routine retrieves the value from the specified key in the USB device's hardware registry.

#### [USBD\\_GetUSBDIVersion](#)

The USBD\_GetUSBDIVersion routine returns version information about the host controller driver (HCD) that controls the client's USB device.Note USBD\_IsInterfaceVersionSupported replaces the USBD\_GetUSBDIVersion routine

#### [USBD\\_IsInterfaceVersionSupported](#)

The USBD\_IsInterfaceVersionSupported routine is called by a USB client driver to check whether the underlying USB driver stack supports a particular USBD interface version.

#### [USBD\\_IsochUrbAllocate](#)

The USBD\_IsochUrbAllocate routine allocates and formats a URB structure for an isochronous transfer request.

#### [USBD\\_ParseConfigurationDescriptor](#)

The USBD\_ParseConfigurationDescriptor routine has been deprecated. Use USBD\_ParseConfigurationDescriptorEx instead.

#### [USBD\\_ParseConfigurationDescriptorEx](#)

The USBD\_ParseConfigurationDescriptorEx routine searches a given configuration descriptor and returns a pointer to an interface that matches the given search criteria.

## [USBD\\_ParseDescriptors](#)

The USBD\_ParseDescriptors routine searches a given configuration descriptor and returns a pointer to the first descriptor that matches the search criteria.

## [USBD\\_QueryBusTime](#)

The USBD\_QueryBusTime routine has been deprecated in Windows XP and later operating systems. Do not use.

## [USBD\\_QueryUsbCapability](#)

The USBD\_QueryUsbCapability routine is called by a WDM client driver to determine whether the underlying USB driver stack and the host controller hardware support a specific capability.

## [USBD\\_RegisterHcFilter](#)

The USBD\_RegisterHcFilter routine has been deprecated in Windows XP and later operating systems.

## [USBD\\_SelectConfigUrbAllocateAndBuild](#)

The USBD\_SelectConfigUrbAllocateAndBuild routine allocates and formats a URB structure that is required to select a configuration for a USB device.

## [USBD\\_SelectInterfaceUrbAllocateAndBuild](#)

The USBD\_SelectInterfaceUrbAllocateAndBuild routine allocates and formats a URB structure that is required for a request to select an interface or change its alternate setting.

## [USBD\\_UrbAllocate](#)

The USBD\_UrbAllocate routine allocates a USB Request Block (URB).

## [USBD\\_UrbFree](#)

The USBD\_UrbFree routine releases the URB that is allocated by USBD\_UrbAllocate, USBD\_IsochUrbAllocate, USBD\_SelectConfigUrbAllocateAndBuild, or USBD\_SelectInterfaceUrbAllocateAndBuild.

## [USBD\\_ValidateConfigurationDescriptor](#)

The USBD\_ValidateConfigurationDescriptor routine validates all descriptors returned by a device in its response to a configuration descriptor request.

## [USBFN\\_GET\\_ATTACH\\_ACTION](#)

The filter driver's implementation that gets invoked when charger is attached to the port.

## [USBFN\\_GET\\_ATTACH\\_ACTION\\_ABORT](#)

The filter driver's implementation to abort an attach-detect operation.

## [USBFN\\_SET\\_DEVICE\\_STATE](#)

The filter driver's implementation to set the device state and operating bus speed.

## [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS\\_INIT](#)

Initializes a **USBPM\_ASSIGN\_CONNECTOR\_POWER\_LEVEL\_PARAMS** structure.

## [UsbPm\\_AssignConnectorPowerLevel](#)

Attempts a PD contract renegotiation with the specified voltage/current/power value.

## [USBPM\\_CLIENT\\_CONFIG\\_EXTRA\\_INFO\\_INIT](#)

Initializes a **USBPM\_CLIENT\_CONFIG\_EXTRA\_INFO** structure.

## [USBPM\\_CLIENT\\_CONFIG\\_INIT](#)

Initializes a **USBPM\_CLIENT\_CONFIG** structure.

## [USBPM\\_CONNECTOR\\_PROPERTIES\\_INIT](#)

Initializes a **USBPM\_CONNECTOR\_PROPERTIES** structure.

## [USBPM\\_CONNECTOR\\_STATE\\_INIT](#)

Initializes a **USBPM\_CONNECTOR\_STATE\_INIT** structure.

## [UsbPm\\_Deregister](#)

Unregisters the client driver with the Policy Manager.

## [USBPM\\_HUB\\_CONNECTOR\\_HANDLES\\_INIT](#)

Initializes a **USBPM\_HUB\_CONNECTOR\_HANDLES** structure.

## [USBPM\\_HUB\\_PROPERTIES\\_INIT](#)

Initializes a [**USBPM\_HUB\_PROPERTIES**] structure.

## [UsbPm\\_Register](#)

Registers the client driver with the Policy Manager to report hub arrival/removal and connector state changes.

## [UsbPm\\_RetrieveConnectorProperties](#)

Retrieves the properties of a connector. The properties are static information that do not change during the lifecycle of a connector.

## [UsbPm\\_RetrieveConnectorState](#)

Retrieves the current state of a connector. Unlike connector properties, state information is dynamic, which can change at runtime.

## [UsbPm\\_RetrieveHubConnectorHandles](#)

Retrieves connector handles for all connectors of a hub.

## [UsbPm\\_RetrieveHubProperties](#)

Retrieves the properties of a hub. Properties are static information that do not change during the lifecycle of a hub.

# Structures

[Expand table](#)

## [\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#)

The \_URB\_BULK\_OR\_INTERRUPT\_TRANSFER structure is used by USB client drivers to send or receive data on a bulk pipe or on an interrupt pipe.

## [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

The \_URB\_CONTROL\_DESCRIPTOR\_REQUEST structure is used by USB client drivers to get or set descriptors on a USB device.

## [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#)

The \_URB\_CONTROL\_FEATURE\_REQUEST structure is used by USB client drivers to set or clear features on a device, interface, or endpoint.

## [\\_URB\\_CONTROL\\_GET\\_CONFIGURATION\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_CONFIGURATION\_REQUEST structure is used by USB client drivers to retrieve the current configuration for a device.

## [\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_INTERFACE\_REQUEST structure is used by USB client drivers to retrieve the current alternate interface setting for an interface in the current configuration.

#### [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_STATUS\_REQUEST structure is used by USB client drivers to retrieve status from a device, interface, endpoint, or other device-defined target.

#### [\\_URB\\_CONTROL\\_TRANSFER](#)

The \_URB\_CONTROL\_TRANSFER structure is used by USB client drivers to transfer data to or from a control pipe.

#### [\\_URB\\_CONTROL\\_TRANSFER\\_EX](#)

The \_URB\_CONTROL\_TRANSFER\_EX structure is used by USB client drivers to transfer data to or from a control pipe, with a timeout that limits the acceptable transfer time.

#### [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#)

The \_URB\_CONTROL\_VENDOR\_OR\_CLASS\_REQUEST structure is used by USB client drivers to issue a vendor or class-specific command to a device, interface, endpoint, or other device-defined target.

#### [\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#)

The \_URB\_GET\_CURRENT\_FRAME\_NUMBER structure is used by USB client drivers to retrieve the current frame number.

#### [\\_URB\\_GET\\_ISOCH\\_PIPE\\_TRANSFER\\_PATH\\_DELAYS](#)

The \_URB\_GET\_ISOCH\_PIPE\_TRANSFER\_PATH\_DELAYS structure is used by USB client drivers to retrieve delays associated with isochronous transfer programming in the host controller and transfer completion so that the client driver can ensure that the device gets the isochronous packets in time.

#### [\\_URB\\_HEADER](#)

The \_URB\_HEADER structure is used by USB client drivers to provide basic information about the request being sent to the host controller driver.

#### [\\_URB\\_ISOCH\\_TRANSFER](#)

The \_URB\_ISOCH\_TRANSFER structure is used by USB client drivers to send data to or retrieve data from an isochronous transfer pipe.

#### [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#)

The \_URB\_OPEN\_STATIC\_STREAMS structure is used by a USB client driver to open streams in the specified bulk endpoint.

#### [\\_URB\\_OS\\_FEATURE\\_DESCRIPTOR\\_REQUEST](#)

The \_URB\_OS\_FEATURE\_DESCRIPTOR\_REQUEST structure is used by the USB hub driver to retrieve Microsoft OS Feature Descriptors from a USB device or an interface on a USB device.

#### [\\_URB\\_PIPE\\_REQUEST](#)

The \_URB\_PIPE\_REQUEST structure is used by USB client drivers to clear a stall condition on an endpoint.

#### [\\_URB\\_SELECT\\_CONFIGURATION](#)

The \_URB\_SELECT\_CONFIGURATION structure is used by client drivers to select a configuration for a USB device.

#### [\\_URB\\_SELECT\\_INTERFACE](#)

The \_URB\_SELECT\_INTERFACE structure is used by USB client drivers to select an alternate setting for an interface or to change the maximum packet size of a pipe in the current configuration on a USB device.

#### [ADDRESS0\\_OWNERSHIP\\_ACQUIRE](#)

Contains parameters for configuring the device.

#### [ALTERNATE\\_INTERFACE](#)

The ALTERNATE\_INTERFACE structure provides information about alternate settings for a Universal Serial Bus (USB) interface.

#### [COMPOSITE\\_DEVICE\\_CAPABILITIES](#)

The COMPOSITE\_DEVICE\_CAPABILITIES structure specifies the capabilities of the driver of a USB multi-function device (composite driver). To initialize the structure, use the COMPOSITE\_DEVICE\_CAPABILITIES\_INIT macro.

#### [CONTROLLER\\_USB\\_20\\_HARDWARE\\_LPM\\_FLAGS](#)

Describes supported protocol capabilities for Link Power Management (LPM) in as defined the USB 2.0 specification.

#### [DEFAULT\\_ENDPOINT\\_UPDATE](#)

Contains the handle to the default endpoint to update in a framework request that is passed by UCX when it invokes EVT\_UCX\_DEFAULT\_ENDPOINT\_UPDATE callback function.

## [ENDPOINT\\_RESET](#)

Describes information required to reset an endpoint. This structure is passed by UCX in the EVT\_UCX\_ENDPOINT\_RESET callback function.

## [ENDPOINTS\\_CONFIGURE](#)

Describes endpoints to enable or disable endpoints. This structure is passed by UCX in the EVT\_UCX\_USBDEVICE\_ENDPOINTS\_CONFIGURE callback function.

## [ENDPOINTS\\_CONFIGURE\\_FAILURE\\_FLAGS](#)

This structure provides failure flags to indicate errors, if any, that might have occurred during a request to an EVT\_UCX\_USBDEVICE\_ENDPOINTS\_CONFIGURE callback function.

## [HUB\\_DEVICE\\_CONFIG\\_INFO](#)

The HUB\_DEVICE\_CONFIG\_INFO structure is used in conjunction with the kernel-mode IOCTL, IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO to request to report information about a USB device and the hub to which the device is attached.

## [HUB\\_INFO\\_FROM\\_PARENT](#)

Describes information about a hub from its parent device.

## [PARENT\\_HUB\\_FLAGS](#)

This structure is used by the HUB\_INFO\_FROM\_PARENT structure to get hub information from the parent.

## [REGISTER\\_COMPOSITE\\_DEVICE](#)

The REGISTER\_COMPOSITE\_DEVICE structure is used with the IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE I/O control request to register a parent driver of a Universal Serial Bus (USB) multi-function device (composite driver) with the USB driver stack.

## [REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#)

The purpose of the REQUEST\_REMOTE\_WAKE\_NOTIFICATION structure is to specify input parameters for the IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION I/O control request.

## [ROOTHUB\\_20PORT\\_INFO](#)

Provides information about a USB 2.0 root hub port. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_20PORT\_INFO callback function.

## [ROOTHUB\\_20PORTS\\_INFO](#)

This structure that has an array of 2.0 ports supported by the root hub. This structure is provided by UCX in a framework request in the EVT\_UCX\_ROOTHUB\_GET\_20PORT\_INFO callback function.

## [ROOTHUB\\_30PORT\\_INFO](#)

Provides information about a USB 3.0 root hub port. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_30PORT\_INFO callback function.

## [ROOTHUB\\_30PORT\\_INFO\\_EX](#)

Provides extended USB 3.0 port information about speed.

## [ROOTHUB\\_30PORTS\\_INFO](#)

Provides information about USB 3.0 root hub ports. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_30PORT\_INFO callback function.

## [ROOTHUB\\_INFO](#)

Provides information about a USB root hub. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_INFO callback function.

## [STREAM\\_INFO](#)

This structure stores information about a stream associated with a bulk endpoint.

## [UCM\\_CONNECTOR\\_CONFIG](#)

Describes the configuration options for a Type-C connector object. An initialized UCM\_MANAGER\_CONFIG structure is an input parameter value to UcmInitializeDevice.

## [UCM\\_CONNECTOR\\_PD\\_CONFIG](#)

Describes the Power Delivery 2.0 capabilities of the connector.

## [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#)

Describes the parameters for PD connection changed event.

## [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#)

Describes the partner that is currently attached to the connector.

## [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG](#)

Describes the configuration options for a Type-C connector.

## [UCM\\_MANAGER\\_CONFIG](#)

Describes the configuration options for the UCM Manager. An initialized UCM\_MANAGER\_CONFIG structure is an input parameter value to `UcmInitializeDevice`.

## [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

Describes a Power Data Object. For information about these members, see the Power Delivery specification.

## [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#)

Describes a Request Data Object (RDO). For information about these members, see the Power Delivery specification.

## [UCMTCPCI\\_DEVICE\\_CONFIG](#)

Used in the client driver's call to `UcmTcpciDeviceInitialize`. Call `UCMTCPCI_DEVICE_CONFIG_INIT` to initialize this structure.

## [UCMTCPCI\\_DRIVER\\_GLOBALS](#)

The global structure for the USB Type-C Port Controller Interface framework extension (`UcmTcpciCx`).

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_DATA](#)

Contains information about hardware alerts received on the port controller object. This structure is used in the `UcmTcpciPortControllerAlert` call. Call `UCMTCPCI_PORT_CONTROLLER_ALERT_DATA_INIT` to initialize this structure.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED\\_IN\\_PARAMS](#)

Stores information about the alternate mode that was detected. This structure is used in the `IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_ENTERED` request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED\\_IN\\_PARAMS](#)

Stores information about the alternate mode that was exited. This structure is used in the `IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_EXITED` request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#)

Contains information about the capabilities of the port controller.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#)

Contains configuration options for the port controller object, passed by the client driver in the call

to UcmTcpciPortControllerCreate. Call UCMTPCI\_PORT\_CONTROLLER\_CONFIG\_INIT to initialize this structure.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED\\_IN\\_PARAMS](#)

Stores information about the pin assignment of the DisplayPort alternate mode that was configured. This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_CONFIGURED request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#)

Stores information about display out status of the DisplayPort connection. This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_DISPLAY\_OUT\_STATUS\_CHANGED request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#)

Stores information about hot plug detect status of the DisplayPort connection. This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_HPD\_STATUS\_CHANGED request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_IN\\_PARAMS](#)

This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_GET\_CONTROL request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_OUT\\_PARAMS](#)

Stores the values of all control registers of the port controller retrieved by the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_GET\_CONTROL request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_IN\\_PARAMS](#)

This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_GET\_STATUS request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_OUT\\_PARAMS](#)

Stores the values of all status registers of the port controller. This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_GET\_STATUS request.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#)

Contains identification information and USB specification version information (in BCD format) about the port controller.

#### [UCMTPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND\\_IN\\_PARAMS](#)

Stores the specified command registers. This structure is used in the IOCTL\_UCMTPCI\_PORT\_CONTROLLER\_SET\_COMMAND request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT\\_IN\\_PARAMS](#)

Stores the value of the CONFIG\_STANDARD\_OUTPUT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONFIG\_STANDARD\_OUTPUT request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL\\_IN\\_PARAMS](#)

Stores the values of all control registers. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONTROL request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO\\_IN\\_PARAMS](#)

Stores the value of the VBUS\_VOLTAGE\_ALARM\_LO\_CFG Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_MESSAGE\_HEADER\_INFO request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT\\_IN\\_PARAMS](#)

Stores the value of the RECEIVE\_DETECT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_RECEIVE\_DETECT request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER\\_IN\\_PARAMS](#)

Stores the value of the TRANSMIT\_BUFFER Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT\_BUFFER request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_IN\\_PARAMS](#)

Stores the values of TRANSMIT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT request.

## [UCMUCSI\\_CONNECTOR\\_INFO](#)

Stores information about connectors that cannot be obtained by sending UCSI commands such as "Get Connector Capability".

## [UCMUCSI\\_DEVICE\\_CONFIG](#)

Configuration structure for UcmUcsiDeviceInitialize.

## [UCMUCSI\\_DRIVER\\_GLOBALS](#)

Reserved for UCMUCSI\_DRIVER\_GLOBALS.

## [UCMUCSI\\_PPM\\_CONFIG](#)

Stores configuration information required to create a Platform Policy Manager (PPM).

## [UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#)

	Contains a UCSI data block for input to IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK.
<a href="#">UCMUCSI_PPM_GET_UCSI_DATA_BLOCK_OUT_PARAMS</a>	Contains a UCSI data block for output to IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK.
<a href="#">UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK_IN_PARAMS</a>	Contains a UCSI data block for input to IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK.
<a href="#">UCSI_ACK_CC_CI_COMMAND</a>	Used in the ACK_CC_CI command. See Table 4-7.
<a href="#">UCSI_ALTERNATE_MODE</a>	Used in GET_ALTERNATE_MODES command. See Table 4-26.
<a href="#">UCSI_BM_POWER_SOURCE</a>	Used in GET_CAPABILITY command. See Bit 15:8 in Table 4-14.
<a href="#">UCSI_CCI</a>	Used in GET_CONNECTOR_CAPABILITY command. See Table 4-16.
<a href="#">UCSI_CONNECTOR_RESET_COMMAND</a>	Used in the CONNECTOR_RESET command. See Table 4-5.
<a href="#">UCSI_CONTROL</a>	Used in the SET_NOTIFICATION_ENABLE command. See Table 4-9.
<a href="#">UCSI_DATA_BLOCK</a>	The data structures for memory locations. See Section 3.
<a href="#">UCSI_GET_ALTERNATE_MODES_COMMAND</a>	Used in the GET_ALTERNATE_MODES command. See Table 4-24.
<a href="#">UCSI_GET_ALTERNATE_MODES_IN</a>	Learn how UCSI_GET_ALTERNATE_MODES_IN is used in the GET_ALTERNATE_MODES command. See Table 4-24.
<a href="#">UCSI_GET_CABLE_PROPERTY_COMMAND</a>	Used in the GET_CABLE_PROPERTY command. See Table 4-37.

## [UCSI\\_GET\\_CABLE\\_PROPERTY\\_IN](#)

Used in the GET\_CABLE\_PROPERTY command. See Table 4-39.

## [UCSI\\_GET\\_CAM\\_SUPPORTED\\_COMMAND](#)

Used in the GET\_CAM\_SUPPORTED command. See Table 4-27.

## [UCSI\\_GET\\_CAM\\_SUPPORTED\\_IN](#)

Learn how UCSI\_GET\_CAM\_SUPPORTED\_IN is used in the GET\_CAM\_SUPPORTED command. See Table 4-27.

## [UCSI\\_GET\\_CAPABILITY\\_IN](#)

Used in the GET\_CAPABILITY command. See Table 4-13.

## [UCSI\\_GET\\_CONNECTOR\\_CAPABILITY\\_COMMAND](#)

Used in the GET\_CONNECTOR\_CAPABILITY command. See Table 4-15.

## [UCSI\\_GET\\_CONNECTOR\\_CAPABILITY\\_IN](#)

Used in the GET\_CONNECTOR\_CAPABILITY command.

## [UCSI\\_GET\\_CONNECTOR\\_STATUS\\_COMMAND](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-40.

## [UCSI\\_GET\\_CONNECTOR\\_STATUS\\_IN](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42.

## [UCSI\\_GET\\_CURRENT\\_CAM\\_COMMAND](#)

Used in the GET\_CURRENT\_CAM command. See Table 4-29.

## [UCSI\\_GET\\_CURRENT\\_CAM\\_IN](#)

Used in the GET\_CURRENT\_CAM command. See Table 4-31.

## [UCSI\\_GET\\_ERROR\\_STATUS\\_COMMAND](#)

Used in the GET\_ERROR\_STATUS command. See Table 4-45

## [UCSI\\_GET\\_ERROR\\_STATUS\\_IN](#)

Used in the GET\_ERROR\_STATUS command. See Table 4-47.

## [UCSI\\_GET\\_PDOS\\_COMMAND](#)

Used in the GET\_PDOS command. See Table 4-34.

## [UCSI\\_GET\\_PDOS\\_IN](#)

Used in the GET\_PDOS command. See Table 4-36.

## [UCSI\\_MESSAGE\\_IN](#)

The MESSAGE IN data structure. See Section 3.4.

## [UCSI\\_MESSAGE\\_OUT](#)

The MESSAGE OUT data structure. See Section 3.5.

## [UCSI\\_SET\\_NEW\\_CAM\\_COMMAND](#)

Used in the SET\_NEW\_CAM command. See Table 4-32.

## [UCSI\\_SET\\_NOTIFICATION\\_ENABLE\\_COMMAND](#)

Learn how UCSI\_SET\_NOTIFICATION\_ENABLE\_COMMAND is used in the SET\_NOTIFICATION\_ENABLE command. See Table 4-9.

## [UCSI\\_SET\\_PDM\\_COMMAND](#)

\_UCSI\_SET\_PDM\_COMMAND is obsolete.

## [UCSI\\_SET\\_PDR\\_COMMAND](#)

Used in the SET\_PDR command. See Table 4-22.

## [UCSI\\_SET\\_POWER\\_LEVEL\\_COMMAND](#)

Used in the SET\_POWER\_LEVEL command. See Table 4-48.

## [UCSI\\_SET\\_UOM\\_COMMAND](#)

Used in the SET\_UOM command. See Table 4-18.

## [UCSI\\_SET\\_UOR\\_COMMAND](#)

Used in the SET\_UOR command. See Table 4-20.

## [UCSI\\_VERSION](#)

The VERSION data structure. See Section 3.1.

## [UCX\\_CONTROLLER\\_ACPI\\_INFORMATION](#)

This structure provides information about an advanced Configuration and power interface (ACPI) USB controller.

## [UCX\\_CONTROLLER\\_CONFIG](#)

This structure configuration data for a USB controller.

## [UCX\\_CONTROLLER\\_PCI\\_INFORMATION](#)

This structure provides information about a PCI USB controller.

## [UCX\\_CONTROLLER\\_RESET\\_COMPLETE\\_INFO](#)

Contains information about the operation to reset the controller. This is used by the client driver in its EVT\_UCX\_CONTROLLER\_RESET callback function.

## [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS](#)

Stores the transport characteristics at relevant points in time. This structure is used in the EVT\_UCX\_CONTROLLER\_GET\_TRANSPORT\_CHARACTERISTICS callback function.

## [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_FLAGS](#)

Defines flags for the transport characteristics changes. This structure is used in the EVT\_UCX\_CONTROLLER\_SET\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_NOTIFICATION callback function.

## [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS](#)

This structure provides a list of UCX default endpoint event callback functions.

## [UCX\\_ENDPOINT\\_CHARACTERISTIC](#)

Stores the characteristics of an endpoint.

## [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS](#)

This structure provides a list of pointers to UCX endpoint event callback functions.

## [UCX\\_ENDPOINT\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#)

Stores the isochronous transfer path delay values.

## [UCX\\_ROOTHUB\\_CONFIG](#)

Contains pointers to event callback functions for creating the root hub by calling

`UcxRootHubCreate`. Initialize this structure by calling `UCX_ROOTHUB_CONFIG_INIT` initialization function (see `Ucxclass.h`).

### [UCX\\_USBDEVICE\\_CHARACTERISTIC](#)

Stores the characteristics of a device.

### [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_PATH\\_DELAY](#)

Learn how `UCX_USBDEVICE_CHARACTERISTIC_PATH_DELAY` stores the isochronous transfer path delay values.

### [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#)

This structure provides a list of UCX USB device event callback functions.

### [UCXUSBDEVICE\\_INFO](#)

Contains information about the USB device. This structure is passed by UCX in the `EVT_UCX_CONTROLLER_USBDEVICE_ADD` event callback function.

### [UDECX\\_ENDPOINTS\\_CONFIGURE\\_PARAMS](#)

Contains the configuration options specified by USB device emulation class extension (UdeCx) to the client driver when the class extension invokes `EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE`.

### [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS](#)

Contains the port numbers to which a virtual USB device is connected. Initialize this structure by calling the `UDECX_USB_DEVICE_PLUG_IN_OPTIONS_INIT` method.

### [UDECX\\_USB\\_DEVICE\\_STATE\\_CHANGE\\_CALLBACKS](#)

Initializes a `UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS` structure with pointers to callback functions that are implemented by a UDE client for a virtual USB device.

### [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS](#)

Contains function pointers to endpoint callback functions implemented by the UDE client driver. Initialize this structure by calling `UDECX_USB_ENDPOINT_CALLBACKS_INIT`.

### [UDECX\\_USB\\_ENDPOINT\\_INIT\\_AND\\_METADATA](#)

Contains the descriptors supported by an endpoint of a virtual USB device.

### [UDECX\\_WDF\\_DEVICE\\_CONFIG](#)

Contains pointers to event callback functions implemented by the UDE client driver for a USB host controller. Initialize this structure by calling UDECX\_WDF\_DEVICE\_CONFIG\_INIT.

#### [UFX\\_DEVICE\\_CALLBACKS](#)

The UFX\_DEVICE\_CALLBACKS structure is used to define then event callback functions supported by the client driver.

#### [UFX\\_DEVICE\\_CAPABILITIES](#)

The UFX\_DEVICE\_CAPABILITIES structure is used USB to define properties of the Universal Serial Bus (USB) device created by the controller.

#### [UFX\\_ENDPOINT\\_CALLBACKS](#)

The UFX\_ENDPOINT\_CALLBACKS structure is used to define then event callback functions supported by the client driver.

#### [UFX\\_HARDWARE\\_FAILURE\\_CONTEXT](#)

The UFX\_HARDWARE\_FAILURE\_CONTEXT structure is used to define controller-specific hardware failure properties.

#### [UFX\\_INTERFACE\\_PROPRIETARY\\_CHARGER](#)

Stores pointers to driver-implemented callback functions for handling proprietary charger operations.

#### [UFX\\_PROPRIETARY\\_CHARGER](#)

Describes the proprietary charger's device power requirements.

#### [URB](#)

The URB structure is used by USB client drivers to describe USB request blocks (URBs) that send requests to the USB driver stack. The URB structure defines a format for all possible commands that can be sent to a USB device.

#### [URS\\_CONFIG](#)

Contains pointers to event callback functions implemented by the URS client driver for a USB dual-role controller. Initialize this structure by calling URS\_CONFIG\_INIT.

#### [USB\\_30\\_HUB\\_DESCRIPTOR](#)

The USB\_30\_HUB\_DESCRIPTOR structure contains a SuperSpeed hub descriptor. For information about the structure members, see Universal Serial Bus Revision 3.0 Specification, 10.13.2.1 Hub Descriptor, Table 10-3. SuperSpeed Hub Descriptor.

## [USB\\_BUS\\_INFORMATION\\_LEVEL\\_0](#)

The USB\_BUS\_INFORMATION\_LEVEL\_0 structure is used in conjunction with the QueryBusInformation interface routine to report information about the bus.

## [USB\\_BUS\\_INFORMATION\\_LEVEL\\_1](#)

The USB\_BUS\_INFORMATION\_LEVEL\_1 structure is used in conjunction with the QueryBusInformation interface routine to report information about the bus.

## [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#)

The USB\_BUS\_INTERFACE\_USBDI\_V0 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## [USB\\_BUS\\_INTERFACE\\_USBDI\\_V1](#)

The USB\_BUS\_INTERFACE\_USBDI\_V1 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## [USB\\_BUS\\_INTERFACE\\_USBDI\\_V2](#)

The USB\_BUS\_INTERFACE\_USBDI\_V2 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#)

The USB\_BUS\_INTERFACE\_USBDI\_V3 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## [USB\\_BUS\\_NOTIFICATION](#)

Learn more about: [\\_USB\\_BUS\\_NOTIFICATION](#) structure

## [USB\\_COMMON\\_DESCRIPTOR](#)

The USB\_COMMON\_DESCRIPTOR structure contains the head of the first descriptor that matches the search criteria in a call to USBD\_ParseDescriptors.

## [USB\\_CONFIGURATION\\_DESCRIPTOR](#)

The USB\_CONFIGURATION\_DESCRIPTOR structure is used by USB client drivers to hold a USB-defined configuration descriptor.

## [USB\\_CYCLE\\_PORT\\_PARAMS](#)

The USB\_CYCLE\_PORT\_PARAMS structure is used with the IOCTL\_USB\_HUB\_CYCLE\_PORT I/O control request to power cycle the port that is associated with the PDO that receives the request.

## [USB\\_DESCRIPTOR\\_REQUEST](#)

The USB\_DESCRIPTOR\_REQUEST structure is used with the IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION I/O control request to retrieve one or more descriptors for the device that is associated with the indicated connection index.

## [USB\\_DEVICE\\_CAPABILITY\\_FIRMWARE\\_STATUS\\_DESCRIPTOR](#)

USB FW Update as defined in the USB 3.2 ENGINEERING CHANGE NOTICE.

## [USB\\_DEVICE\\_CHARACTERISTICS](#)

Contains information about the USB device's characteristics, such as the maximum send and receive delays for any request. This structure is used in the IOCTL\_USB\_GET\_DEVICE\_CHARACTERISTICS request.

## [USB\\_DEVICE\\_DESCRIPTOR](#)

The USB\_DEVICE\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined device descriptor.

## [USB\\_DEVICE\\_PORT\\_PATH](#)

Contains the port path of a USB device.

## [USB\\_DEVICE\\_QUALIFIER\\_DESCRIPTOR](#)

The USB\_DEVICE\_QUALIFIER\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined device qualifier descriptor.

## [USB\\_ENDPOINT\\_DESCRIPTOR](#)

The USB\_ENDPOINT\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined endpoint descriptor.

## [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

Stores the frame and microframe numbers and the calculated system QPC values. This structure is used in the IOCTL\_USB\_GET\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC request.

## [USB\\_HCD\\_DRIVERKEY\\_NAME](#)

The USB\_HCD\_DRIVERKEY\_NAME structure is used with the IOCTL\_GET\_HCD\_DRIVERKEY\_NAME I/O control request to retrieve the driver key in the registry for the USB host controller driver.

## [USB\\_HUB\\_CAP\\_FLAGS](#)

The USB\_HUB\_CAP\_FLAGS structure is used to report the capabilities of a hub.

## [USB\\_HUB\\_CAPABILITIES](#)

The USB\_HUB\_CAPABILITIES structure has been deprecated. Use USB\_HUB\_CAPABILITIES\_EX instead.

## [USB\\_HUB\\_CAPABILITIES\\_EX](#)

The USB\_HUB\_CAPABILITIES\_EX structure is used with the IOCTL\_USB\_GET\_HUB\_CAPABILITIES I/O control request to retrieve the capabilities of a particular USB hub.

## [USB\\_HUB\\_DESCRIPTOR](#)

The USB\_HUB\_DESCRIPTOR structure contains a hub descriptor.

## [USB\\_HUB\\_INFORMATION](#)

The USB\_HUB\_INFORMATION structure contains information about a hub.

## [USB\\_HUB\\_INFORMATION\\_EX](#)

The USB\_HUB\_INFORMATION\_EX structure is used with the IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX I/O control request to retrieve information about a Universal Serial Bus (USB) hub.

## [USB\\_HUB\\_NAME](#)

The USB\_HUB\_NAME structure stores the hub's symbolic device name.

## [USB\\_ID\\_STRING](#)

The USB\_ID\_STRING structure is used to store a string or multi-string.

## [USB\\_INTERFACE\\_DESCRIPTOR](#)

The USB\_INTERFACE\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined interface descriptor.

## [USB\\_MI\\_PARENT\\_INFORMATION](#)

The USB\_MI\_PARENT\_INFORMATION structure contains information about a composite device.

## [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

The USB\_NODE\_CONNECTION\_ATTRIBUTES structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES I/O control request to retrieve the attributes of a connection.

## [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)

The USB\_NODE\_CONNECTION\_DRIVERKEY\_NAME structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME I/O control request to retrieve the driver key name for the device that is connected to the indicated port.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION](#)

The USB\_NODE\_CONNECTION\_INFORMATION structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION request to retrieve information about a USB port and connected device.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX structure is used in conjunction with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request to obtain information about the connection associated with the indicated USB port.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 I/O control request to retrieve speed information about a Universal Serial Bus (USB) device that is attached to a particular port.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2\\_FLAGS](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2\_FLAGS union is used to indicate the speed at which a USB 3.0 device is currently operating and whether it can operate at higher speed, when attached to a particular port.

#### [USB\\_NODE\\_CONNECTION\\_NAME](#)

The USB\_NODE\_CONNECTION\_NAME structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME I/O control request to retrieve the symbolic link of the downstream hub that is attached to the port.

#### [USB\\_NODE\\_CONNECTION\\_SUPERSPEEDPLUS\\_INFORMATION](#)

#### [USB\\_NODE\\_INFORMATION](#)

The USB\_NODE\_INFORMATION structure is used with the IOCTL\_USB\_GET\_NODE\_INFORMATION I/O control request to retrieve information about a parent device.

#### [USB\\_PIPE\\_INFO](#)

The USB\_PIPE\_INFO structure is used in conjunction with the USB\_NODE\_CONNECTION\_INFORMATION\_EX structure and the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request to obtain information about a connection and its associated pipes.

## [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

The USB\_PORT\_CONNECTOR\_PROPERTIES structure is used with the IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request to retrieve information about a port on a particular SuperSpeed hub.

## [USB\\_PORT\\_PROPERTIES](#)

The USB\_PORT\_PROPERTIES union is used to report the capabilities of a Universal Serial Bus (USB) port. The port capabilities are retrieved in the USB\_PORT\_CONNECTOR\_PROPERTIES structure by the IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request.

## [USB\\_PROTOCOLS](#)

The USB\_PROTOCOLS union is used to report the Universal Serial Bus (USB) signaling protocols that are supported by the port.

## [USB\\_ROOT\\_HUB\\_NAME](#)

The USB\_ROOT\_HUB\_NAME structure stores the root hub's symbolic device name.

## [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

The input and output buffer for the IOCTL\_USB\_START\_TRACKING\_FOR\_TIME\_SYNC request.

## [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

The input buffer for the IOCTL\_USB\_STOP\_TRACKING\_FOR\_TIME\_SYNC request.

## [USB\\_STRING\\_DESCRIPTOR](#)

The USB\_STRING\_DESCRIPTOR structure is used by USB client drivers to hold a USB-defined string descriptor.

## [USB\\_SUPERSPEED\\_ENDPOINT\\_COMPANION\\_DESCRIPTOR](#)

The USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined SuperSpeed Endpoint Companion descriptor. For more information, see section 9.6.7 and Table 9-20 in the official USB 3.0 specification.

## [USB\\_TOPOLOGY\\_ADDRESS](#)

The USB\_TOPOLOGY\_ADDRESS structure is used with the IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS I/O request to retrieve information about a USB device's location in the USB device tree.

## [USB\\_TRANSPORT\\_CHARACTERISTICS](#)

Stores the transport characteristics at relevant points in time. This structure is used in the IOCTL\_USB\_GET\_TRANSPORT\_CHARACTERISTICS request.

#### [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#)

Contains registration information filled when the IOCTL\_USB\_REGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request completes.

#### [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_REGISTRATION](#)

Contains registration information for the IOCTL\_USB\_REGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request.

#### [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_UNREGISTRATION](#)

Contains unregistration information for the IOCTL\_USB\_UNREGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request.

#### [USBC\\_DEVICE\\_CONFIGURATION\\_INTERFACE\\_V1](#)

The USBC\_DEVICE\_CONFIGURATION\_INTERFACE\_V1 structure is exposed by the vendor-supplied filter drivers to assist the USB generic parent driver in defining interface collections.

#### [USBC\\_FUNCTION\\_DESCRIPTOR](#)

The USBC\_FUNCTION\_DESCRIPTOR structure describes a USB function and its associated interface collection.

#### [USBC\\_PD\\_ALTERNATE\\_MODE](#)

Stores information about the alternate mode that was detected.

#### [USBC\\_PD\\_POWER\\_DATA\\_OBJECT](#)

Describes a power data object (PDO).

#### [USBC\\_PD\\_REQUEST\\_DATA\\_OBJECT](#)

Describes a request data object (RDO).

#### [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION](#)

Stores xHCI-specific V2 information that is used by client drivers to transfer data to and from the offloaded endpoints.

#### [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION\\_V1](#)

Stores xHCI-specific V1 information that is used by client drivers to transfer data to and from the offloaded endpoints.

## [USBD\\_INTERFACE\\_INFORMATION](#)

The USBD\_INTERFACE\_INFORMATION structure holds information about an interface for a configuration on a USB device.

## [USBD\\_INTERFACE\\_LIST\\_ENTRY](#)

The USBD\_INTERFACE\_LIST\_ENTRY structure is used by USB client drivers to create an array of interfaces to be inserted into a configuration request.

## [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#)

The USBD\_ISO\_PACKET\_DESCRIPTOR structure is used by USB client drivers to describe an isochronous transfer packet.

## [USBD\\_PIPE\\_INFORMATION](#)

The USBD\_PIPE\_INFORMATION structure is used by USB client drivers to hold information about a pipe from a specific interface.

## [USBD\\_STREAM\\_INFORMATION](#)

The USBD\_STREAM\_INFORMATION structure stores information about a stream associated with a bulk endpoint.

## [USBD\\_VERSION\\_INFORMATION](#)

The USBD\_VERSION\_INFORMATION structure is used by the GetUSBDIVersion function to report its output data.

## [USBDEVICE\\_ABORTIO](#)

Contains a handle for the Universal Serial Bus (USB) hub or device for which to abort data transfers.

## [USBDEVICE\\_ADDRESS](#)

Contains parameters for a request to transition the specified device to the Addressed state. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_ADDRESS callback function.

## [USBDEVICE\\_DISABLE](#)

Contains parameters for a request to disable the specified device. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_DISABLE callback function.

## [USBDEVICE\\_ENABLE](#)

Contains parameters for a request to enable the specified device. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_ENABLE callback function.

#### [USBDEVICE\\_ENABLE\\_FAILURE\\_FLAGS](#)

The flags that are set by the client driver in the EVT\_UCX\_USBDEVICE\_ENABLE callback function. Indicate errors, if any, that might have occurred while enabling the device.

#### [USBDEVICE\\_HUB\\_INFO](#)

Contains parameters for a request to get information about the specified hub. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_HUB\_INFO callback function.

#### [USBDEVICE\\_MGMT\\_HEADER](#)

This structure provides a handle for the Universal Serial Bus (USB) hub or device physically connected to the bus.

#### [USBDEVICE\\_PURGEIO](#)

The USBDEVICE\_PURGEIO structure contains the handle for the Universal Serial Bus (USB) hub or device to purge I/O for.

#### [USBDEVICE\\_RESET](#)

Contains parameters for a request to reset the specified device. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_RESET callback function.

#### [USBDEVICE\\_STARTIO](#)

Contains a handle for the Universal Serial Bus (USB) hub or device on which to start data transfer.

#### [USBDEVICE\\_TREE\\_PURGEIO](#)

This structure provides the handle for the Universal Serial Bus (USB) device tree to purge I/O for.

#### [USBDEVICE\\_UPDATE](#)

Passed by UCX to update the specified device. This structure is in the request parameters (Parameters.Others.Arg1) of a framework request object passed in the EVT\_UCX\_USBDEVICE\_UPDATE callback function.

#### [USBDEVICE\\_UPDATE\\_20\\_HARDWARE\\_LPM\\_PARAMETERS](#)

Contains parameters for a request to update USB 2.0 link power management (LPM). UCX passes this structure in the EVT\_UCX\_USBDEVICE\_UPDATE callback function.

## [USBDEVICE\\_UPDATE\\_FAILURE\\_FLAGS](#)

The flags that are set by the client driver in the EVT\_UCX\_USBDEVICE\_UPDATE callback function. Indicate errors, if any, that might have occurred while updating the device.

## [USBDEVICE\\_UPDATE\\_FLAGS](#)

Contains request flags set by UCX that is passed in the USBDEVICE\_UPDATE structure when UCX invokes the client driver's EVT\_UCX\_USBDEVICE\_UPDATE callback function.

## [USBFN\\_BUS\\_CONFIGURATION\\_INFO](#)

Configuration packet that stores information about an available USB configuration.

## [USBFN\\_CLASS\\_INFORMATION\\_PACKET](#)

Describes device interface class information associated with a USB interface. This structure can only hold information about a single function interface.

## [USBFN\\_CLASS\\_INFORMATION\\_PACKET\\_EX](#)

Describes device interface class information associated with a USB interface. This structure can be used to describe single and multi-interface functions.

## [USBFN\\_CLASS\\_INTERFACE](#)

Describes an interface and its endpoints.

## [USBFN\\_CLASS\\_INTERFACE\\_EX](#)

Learn how USBFN\_CLASS\_INTERFACE\_EX describes an interface and its endpoints.

## [USBFN\\_INTERFACE\\_ATTACH](#)

Stores pointers to driver-implemented callback functions for handling attach and detach operations.

## [USBFN\\_INTERFACE\\_INFO](#)

Learn how USBFN\_INTERFACE\_INFO describes an interface and its endpoints.

## [USBFN\\_NOTIFICATION](#)

Describes information about a Universal Serial Bus (USB) event notification that was received by using IOCTL\_INTERNAL\_USBFN\_BUS\_EVENT\_NOTIFICATION.

## [USBFN\\_ON\\_ATTACH](#)

Describes the detected port type and attach action.

<a href="#">USBFN_PIPE_INFORMATION</a>	Describes attributes of a pipe associated with an endpoint on a specific interface.
<a href="#">USBFN_POWER_FILTER_STATE</a>	Reserved. Do not use.
<a href="#">USBFN_USB_STRING</a>	Describes a USB string descriptor and the associated string index.
<a href="#">USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS</a>	Describes the parameters for the <b>UsbPm_AssignConnectorPowerLevel</b> .
<a href="#">USBPM_CLIENT_CONFIG</a>	The configuration structure used in the registering the client driver with the Policy Manager
<a href="#">USBPM_CLIENT_CONFIG_EXTRA_INFO</a>	Contains optional information used to configure the client driver's registration.
<a href="#">USBPM_CONNECTOR_PROPERTIES</a>	Describes the properties of a connector.
<a href="#">USBPM_CONNECTOR_STATE</a>	Describes the state of a connector.
<a href="#">USBPM_EVENT_CALLBACK_PARAMS</a>	Contains the details of the events related to changes in policy manager arrival/removal, hub arrival/removal or connector state change.
<a href="#">USBPM_HUB_CONNECTOR_HANDLES</a>	Stores the connector handles for all connectors on a hub.
<a href="#">USBPM_HUB_PROPERTIES</a>	Properties of a connector hub.

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# ucmmanager.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

ucmmanager.h contains the following programming interfaces:

## Functions

<a href="#">UCM_CONNECTOR_CONFIG_INIT</a>
Initializes a UCM_CONNECTOR_CONFIG structure.
<a href="#">UCM_CONNECTOR_PD_CONFIG_INIT</a>
Initializes a UCM_CONNECTOR_PD_CONFIG structure.
<a href="#">UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS_INIT</a>
Initializes a UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS structure.
<a href="#">UCM_CONNECTOR_TYPEC_ATTACH_PARAMS_INIT</a>
Initializes a UCM_CONNECTOR_TYPEC_ATTACH_PARAMS structure.
<a href="#">UCM_CONNECTOR_TYPEC_CONFIG_INIT</a>
Initializes the UCM_CONNECTOR_TYPEC_CONFIG structure.
<a href="#">UCM_MANAGER_CONFIG_INIT</a>
Initializes a UCM_MANAGER_CONFIG structure.
<a href="#">UcmConnectorChargingStateChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the updated charging state of the partner connector.
<a href="#">UcmConnectorCreate</a>
Creates a connector object.
<a href="#">UcmConnectorDataDirectionChanged</a>

Notifies the USB connector manager framework extension (UcmCx) with the new data role of a change in data role.
<a href="#">UcmConnectorPdConnectionStateChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the connection capabilities of the currently negotiated PD contract (if any).
<a href="#">UcmConnectorPdPartnerSourceCaps</a>
Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the partner connector.
<a href="#">UcmConnectorPdSourceCaps</a>
Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the connector.
<a href="#">UcmConnectorPowerDirectionChanged</a>
Notifies the USB connector manager framework extension (UcmCx) with the new power role of the partner connector.
<a href="#">UcmConnectorTypeCAttach</a>
Notifies the USB connector manager framework extension (UcmCx) when a partner connector is attached.
<a href="#">UcmConnectorTypeCCurrentAdChanged</a>
Notifies the USB connector manager framework extension (UcmCx) when the specified connector changes the current advertisement. Either the connector changes it (when it is DFP/Source), or the partner changed it (when it is UFP/Sink).
<a href="#">UcmConnectorTypeCDetach</a>
Notifies the USB connector manager framework extension (UcmCx) when the partner connector detaches from the specified Type-C connector.
<a href="#">UcmInitializeDevice</a>
Initializes the USB connector manager framework extension (UcmCx).

## Callback functions

### [EVT\\_UCM\\_CONNECTOR\\_SET\\_DATA\\_ROLE](#)

The client driver's implementation of the EVT\_UCM\_CONNECTOR\_SET\_DATA\_ROLE event callback function that swaps the data role of the connector to the specified role when attached to a partner connector.

### [EVT\\_UCM\\_CONNECTOR\\_SET\\_POWER\\_ROLE](#)

The client driver's implementation of the EVT\_UCM\_CONNECTOR\_SET\_POWER\_ROLE event callback function that sets the power role of the connector to the specified role when attached to a partner connector.

## Structures

### [UCM\\_CONNECTOR\\_CONFIG](#)

Describes the configuration options for a Type-C connector object. An initialized UCM\_MANAGER\_CONFIG structure is an input parameter value to UcmInitializeDevice.

### [UCM\\_CONNECTOR\\_PD\\_CONFIG](#)

Describes the Power Delivery 2.0 capabilities of the connector.

### [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#)

Describes the parameters for PD connection changed event.

### [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#)

Describes the partner that is currently attached to the connector.

### [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG](#)

Describes the configuration options for a Type-C connector.

### [UCM\\_MANAGER\\_CONFIG](#)

Describes the configuration options for the UCM Manager. An initialized UCM\_MANAGER\_CONFIG structure is an input parameter value to UcmInitializeDevice.

# EVT\_UCM\_CONNECTOR\_SET\_DATA\_ROLE callback function (ucmmanager.h)

Article02/22/2024

The client driver's implementation of the *EVT\_UCM\_CONNECTOR\_SET\_DATA\_ROLE* event callback function that swaps the data role of the connector to the specified role when attached to a partner connector.

## Syntax

C++

```
EVT_UCM_CONNECTOR_SET_DATA_ROLE EvtUcmConnectorSetDataRole;

NTSTATUS EvtUcmConnectorSetDataRole(
    [in] UCMCONNECTOR Connector,
    [in] UCM_DATA_ROLE DataRole
)
{...}
```

## Parameters

[in] Connector

Handle to the connector that the client driver received in a previous call to the [UcmConnectorCreate](#) method.

[in] DataRole

A [UCM\\_TYPEC\\_PARTNER](#)-typed flag that specifies the role to set.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To register an *EVT\_UCM\_CONNECTOR\_SET\_DATA\_ROLE* callback function, the client driver must call [UcmConnectorCreate](#).

The USB connector manager framework extension (UcmCx) can request either **UcmTypeCPortStateUfp** or **UcmTypeCPortStateDfp**. If the port is already in the requested role, the client driver can complete the request without any changes. Otherwise, it starts a data-role swap operation (DR\_Swap). The driver calls [UcmConnectorDataDirectionChanged](#) to notify UcmCx about the success or failure of that operation. The driver can call that method within the callback function.

The role persists for the current connection.

If a role-swap operation is pending, UcmCx does not request another role swap. Those operations are serialized across power and data role swaps.

After the swap operation completes, if the partner port sends a DR\_Swap request, the client driver must reject the request.

## Examples

```
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;
}
```

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# EVT\_UCM\_CONNECTOR\_SET\_POWER\_R OLE callback function (ucmmanager.h)

Article10/21/2021

The client driver's implementation of the *EVT\_UCM\_CONNECTOR\_SET\_POWER\_ROLE* event callback function that sets the power role of the connector to the specified role when attached to a partner connector.

## Syntax

C++

```
EVT_UCM_CONNECTOR_SET_POWER_ROLE EvtUcmConnectorSetPowerRole;

NTSTATUS EvtUcmConnectorSetPowerRole(
    [in] UCMCONNECTOR Connector,
    [in] UCM_POWER_ROLE PowerRole
)
{...}
```

## Parameters

[in] Connector

Handle to the connector that the client driver received in a previous call to the [UcmConnectorCreate](#) method.

[in] PowerRole

A [UCM\\_POWER\\_ROLE](#)-typed flag that specifies the role to set.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To register an *EVT\_UCM\_CONNECTOR\_SET\_POWER\_ROLE* callback function, the client must call [UcmConnectorCreate](#).

The USB connector manager framework extension (UcmCx) can request either **UcmPowerRoleSink** or **UcmPowerRoleSource**. If the port is already in the requested role, the client driver can complete the request without any changes. Otherwise, it starts a power-role swap operation (PR\_Swap). The driver calls [UcmConnectorPowerDirectionChanged](#) to notify UcmCx about the success or failure of that operation. The driver can call that method within the callback function.

The role persists for the current connection.

If a role-swap operation is pending, UcmCx does not request another role swap. Those operations are serialized across power and data role swaps.

After the swap operation completes, if the partner port sends a PR\_Swap request, the client driver must reject the request.

## Examples

```
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;
}
```

## Requirements

<b>Minimum supported client</b>	Windows 10
<b>Minimum supported server</b>	Windows Server 2016
<b>Target Platform</b>	Windows
<b>Minimum KMDF version</b>	1.15
<b>Minimum UMDF version</b>	2.15
<b>Header</b>	ucmmanager.h (include Ucmcx.h)
<b>IRQL</b>	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# UCM\_CONNECTOR\_CONFIG structure (ucmmanager.h)

Article 02/22/2024

Describes the configuration options for a Type-C connector object. An initialized **UCM\_MANAGER\_CONFIG** structure is an input parameter value to [UcmInitializeDevice](#).

## Syntax

C++

```
typedef struct _UCM_CONNECTOR_CONFIG {
    ULONG                     Size;
    ULONGLONG                 ConnectorId;
    PUCM_CONNECTOR_TYPEC_CONFIG TypeCConfig;
    PUCM_CONNECTOR_PD_CONFIG   PdConfig;
} UCM_CONNECTOR_CONFIG, *PUCM_CONNECTOR_CONFIG;
```

## Members

**Size**

Size of the **UCM\_CONNECTOR\_CONFIG** structure.

**ConnectorId**

Connector identifier.

**TypeCConfig**

A pointer to an initialized **UCM\_CONNECTOR\_TYPEC\_CONFIG** structure that contains the configuration options for the connector.

**PdConfig**

A pointer to an initialized **UCM\_CONNECTOR\_PD\_CONFIG** structure that contains the power roles supported by the connector.

## Remarks

Initialize this structure by calling [UCM\\_CONNECTOR\\_CONFIG\\_INIT](#). An initialized **UCM\_CONNECTOR\_CONFIG** structure is an input parameter value to [UcmConnectorCreate](#) that is used by the client driver to create a connector object.

## Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

- [UcmConnectorCreate](#)

# UCM\_CONNECTOR\_CONFIG\_INIT function (ucmmanager.h)

Article02/22/2024

Initializes a [UCM\\_CONNECTOR\\_CONFIG](#) structure.

## Syntax

C++

```
void UCM_CONNECTOR_CONFIG_INIT(
    [out] PUCM_CONNECTOR_CONFIG Config,
    [in]   ULONGLONG           ConnectorId
);
```

## Parameters

[out] Config

Pointer to a caller-allocated [UCM\\_CONNECTOR\\_CONFIG](#) structure to initialize.

[in] ConnectorId

The identifier to assign to the connector object. If there is only one connector, pass 0.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Requirement	Value
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UCM\\_MANAGER\\_CONFIG](#)

# UCM\_CONNECTOR\_PD\_CONFIG structure (ucmmanager.h)

Article04/01/2021

Describes the Power Delivery 2.0 capabilities of the connector.

## Syntax

C++

```
typedef struct _UCM_CONNECTOR_PD_CONFIG {
    ULONG             Size;
    BOOLEAN           IsSupported;
    ULONG             SupportedPowerRoles;
    PFN_UCM_CONNECTOR_SET_POWER_ROLE EvtSetPowerRole;
} UCM_CONNECTOR_PD_CONFIG, *PUCM_CONNECTOR_PD_CONFIG;
```

## Members

Size

Size of the UCM\_CONNECTOR\_PD\_CONFIG structure.

IsSupported

If TRUE, a PD role is supported. (Default).

If FALSE, a PD role is not supported.

SupportedPowerRoles

Indicates the operating mode of the connector. This value is a bitwise OR of UCM\_POWER\_ROLE-typed flags.

EvtSetPowerRole

A pointer to the Policy Manager's implementation of the EVT\_UCM\_CONNECTOR\_SET\_POWER\_ROLE event callback.

## Remarks

Initialize this structure by calling [UCM\\_CONNECTOR\\_PD\\_CONFIG\\_INIT](#). An initialized [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG](#) structure is set to the **PdConfig** member of the [UCM\\_CONNECTOR\\_CONFIG](#) structure, which is an input parameter value to [UcmConnectorCreate](#) that is called by Policy Manager to create a connector object.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorCreate](#)

# UCM\_CONNECTOR\_PD\_CONFIG\_INIT function (ucmmanager.h)

Article02/22/2024

Initializes a [UCM\\_CONNECTOR\\_PD\\_CONFIG](#) structure.

## Syntax

C++

```
void UCM_CONNECTOR_PD_CONFIG_INIT(
    [out] PUCM_CONNECTOR_PD_CONFIG Config,
    [in]   ULONG           SupportedPowerRoles
);
```

## Parameters

[out] [Config](#)

Pointer to a caller-allocated [UCM\\_CONNECTOR\\_PD\\_CONFIG](#) structure to initialize.

[in] [SupportedPowerRoles](#)

A bitwise OR of [UCM\\_POWER\\_ROLE](#)-typed flags.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Requirement	Value
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorCreate](#)

# UCM\_CONNECTOR\_PD\_CONN\_STATE\_C HANGED\_PARAMS structure (ucmmanager.h)

Article02/22/2024

Describes the parameters for PD connection changed event.

## Syntax

C++

```
typedef struct _UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS {
    ULONG             Size;
    UCM_PD_CONN_STATE PdConnState;
    UCM_PD_REQUEST_DATA_OBJECT Rdo;
    UCM_CHARGING_STATE ChargingState;
} UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS,
*PUCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS;
```

## Members

Size

Size of the UCM\_CONNECTOR\_PD\_CONN\_STATE\_CHANGED\_PARAMS structure.

PdConnState

The state of the connector indicated by one of the [UCM\\_PD\\_CONN\\_STATE](#)-typed flags.

Rdo

An initialized [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#) structure that describes the characteristics of the new connection state.

ChargingState

Charging state of the port indicated by one of the [UCM\\_CHARGING\\_STATE](#)-typed flags.

## Remarks

Initialize this structure by calling [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS\\_INIT](#). An initialized [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#) structure is an input parameter value to [UcmConnectorPdConnectionStateChanged](#) that is used by the client driver to notify UcmCx about the Attached state of the port.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorPdConnectionStateChanged](#)

[UcmConnectorTypeCAttach](#)

# UCM\_CONNECTOR\_PD\_CONN\_STATE\_CHANGED\_PARAMS\_INIT function (ucmmanager.h)

Article02/22/2024

Initializes a [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#) structure.

## Syntax

C++

```
void UCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS_INIT(
    [out] PUCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS Params,
    [in]   UCM_PD_CONN_STATE                      PdConnState
);
```

## Parameters

[out] Params

Pointer to a caller-allocated [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#) structure to initialize.

[in] PdConnState

A [UCM\\_PD\\_CONN\\_STATE](#)-typed flag that indicates the connection state of the partner port.

## Return value

None

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10

<b>Requirement</b>	<b>Value</b>
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorPdConnectionStateChanged](#)

# UCM\_CONNECTOR\_TYPEC\_ATTACH\_PARAMS structure (ucmmanager.h)

Article02/22/2024

Describes the partner that is currently attached to the connector.

## Syntax

C++

```
typedef struct _UCM_CONNECTOR_TYPEC_ATTACH_PARAMS {
    ULONG             Size;
    UCM_TYPEC_PARTNER Partner;
    UCM_TYPEC_CURRENT CurrentAdvertisement;
    UCM_CHARGING_STATE ChargingState;
} UCM_CONNECTOR_TYPEC_ATTACH_PARAMS, *PUCM_CONNECTOR_TYPEC_ATTACH_PARAMS;
```

## Members

Size

Size of the UCM\_CONNECTOR\_TYPEC\_ATTACH\_PARAMS structure.

Partner

The type of partner attached to the connector, indicated by a [UCM\\_TYPEC\\_PARTNER](#) value.

CurrentAdvertisement

Power sourcing capabilities of: the partner port when [PortPartnerType](#) is [UcmTypeCPortStateDfp](#); the local port when [PortPartnerType](#) is not [UcmTypeCPortStateDfp](#).

ChargingState

Optional. Charging state of the port indicated by one of the [UCM\\_CHARGING\\_STATE](#)-typed flags.

## Remarks

Initialize this structure by calling [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS\\_INIT](#). An initialized **UCM\_CONNECTOR\_TYPEC\_ATTACH\_PARAMS** structure is an input parameter value to [UcmConnectorTypeCAttach](#) that is used by the client driver to notify UcmCx about the Attached state of the port.

## Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

- [UcmConnectorTypeCAttach](#)

# UCM\_CONNECTOR\_TYPEC\_ATTACH\_PARAMS\_INIT function (ucmmanager.h)

Article 02/22/2024

Initializes a [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#) structure.

## Syntax

C++

```
void UCM_CONNECTOR_TYPEC_ATTACH_PARAMS_INIT(
    PUCM_CONNECTOR_TYPEC_ATTACH_PARAMS Params,
    UCM_TYPEC_PARTNER                 Partner
);
```

## Parameters

Params

Pointer to a caller-allocated [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#) structure to initialize.

Partner

A [UCM\\_TYPE\\_C\\_PORT\\_STATE](#)-typed flag that indicates the state of the partner port.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorTypeCAttach](#)

# UCM\_CONNECTOR\_TYPEC\_CONFIG structure (ucmmanager.h)

Article02/22/2024

Describes the configuration options for a Type-C connector.

## Syntax

C++

```
typedef struct _UCM_CONNECTOR_TYPEC_CONFIG {
    ULONG                     Size;
    BOOLEAN                   IsSupported;
    ULONG                     SupportedOperatingModes;
    ULONG                     SupportedPowerSourcingCapabilities;
    BOOLEAN                   AudioAccessoryCapable;
    PFN_UCM_CONNECTOR_SET_DATA_ROLE EvtSetDataRole;
} UCM_CONNECTOR_TYPEC_CONFIG, *PUCM_CONNECTOR_TYPEC_CONFIG;
```

## Members

Size

Size of the **UCM\_CONNECTOR\_TYPEC\_CONFIG** structure.

IsSupported

TRUE indicates a Type-C connector. FALSE, otherwise. is supported.

SupportedOperatingModes

Indicates the supported operating mode of the connector. This value is a bitwise OR of **UCM\_TYPEC\_OPERATING\_MODE**-typed flags.

SupportedPowerSourcingCapabilities

Indicates the supported power source capabilities of the connector. This value is a bitwise OR of **UCM\_TYPEC\_CURRENT**-typed flags.

AudioAccessoryCapable

Indicates whether the connector is capable of detecting a USB Type-C analog input as 3.5 mm audio jack.

### EvtSetDataRole

A pointer to the client driver's implementation of the [EVT\\_UCM\\_CONNECTOR\\_SET\\_DATA\\_ROLE](#) callback function.

## Remarks

Initialize this structure by calling [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG\\_INIT](#). An initialized UCM\_CONNECTOR\_TYPEC\_CONFIG structure is an input parameter value to [UcmConnectorCreate](#) that is used by Policy Manager to create a connector object.

## Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UcmConnectorCreate](#)

# UCM\_CONNECTOR\_TYPEC\_CONFIG\_INIT function (ucmmanager.h)

Article 02/22/2024

Initializes the [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG](#) structure.

## Syntax

C++

```
void UCM_CONNECTOR_TYPEC_CONFIG_INIT(
    [out] PUCM_CONNECTOR_TYPEC_CONFIG Config,
    [in]  ULONG                      SupportedOperatingModes,
    [in]  ULONG                      SupportedPowerSourcingCapabilities
);
```

## Parameters

[out] Config

Pointer to a caller-allocated [UCM\\_CONNECTOR\\_TYPEC\\_CONFIG](#) structure to initialize.

[in] SupportedOperatingModes

Indicates the operating mode of the connector. This value is a bitwise OR of [UCM\\_TYPEC\\_OPERATING\\_MODE](#)-typed flags.

[in] SupportedPowerSourcingCapabilities

Indicates the power source capabilities of the connector. This value is a bitwise OR of [UCM\\_TYPEC\\_CURRENT](#)-typed flags.

## Return value

None

## Requirements

[+] [Expand table](#)

<b>Requirement</b>	<b>Value</b>
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

# UCM\_MANAGER\_CONFIG structure (ucmmanager.h)

Article02/22/2024

Describes the configuration options for the UCM Manager. An initialized **UCM\_MANAGER\_CONFIG** structure is an input parameter value to [UcmInitializeDevice](#).

## Syntax

C++

```
typedef struct _UCM_MANAGER_CONFIG {
    ULONG Size;
} UCM_MANAGER_CONFIG, *PUCM_MANAGER_CONFIG;
```

## Members

Size

Size of the **UCM\_MANAGER\_CONFIG** structure. Initialize this structure by calling [UCM\\_MANAGER\\_CONFIG\\_INIT](#).

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

UCM\_MANAGER\_CONFIG\_INIT

# UCM\_MANAGER\_CONFIG\_INIT function (ucmmanager.h)

Article02/22/2024

Initializes a [UCM\\_MANAGER\\_CONFIG](#) structure.

## Syntax

C++

```
void UCM_MANAGER_CONFIG_INIT(
    [out] PUCM_MANAGER_CONFIG Config
);
```

## Parameters

[out] Config

Pointer to a caller-allocated [UCM\\_MANAGER\\_CONFIG](#) structure to initialize.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)

## See also

[UCM\\_MANAGER\\_CONFIG](#)

# UcmConnectorChargingStateChanged function (ucmmanager.h)

Article02/22/2024

Notifies the USB connector manager framework extension (UcmCx) with the updated charging state of the partner connector.

## Syntax

C++

```
NTSTATUS UcmConnectorChargingStateChanged(
    [in] UCMCONNECTOR      Connector,
    [in] UCM_CHARGING_STATE ChargingState
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] ChargingState

One of the [UCM\\_CHARGING\\_STATE](#)-typed flags that indicates the new charging state.

## Return value

[UcmConnectorChargingStateChanged](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

If the client driver determines that the charging state is non-optimal, it should report slow or trickle charging to UcmCx. Then, the operating system notifies the user of this condition.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# UcmConnectorCreate function (ucmmanager.h)

Article 10/21/2021

Creates a connector object.

## Syntax

C++

```
NTSTATUS UcmConnectorCreate(
    [in] WDFDEVICE             WdfDevice,
    [in] PUCM_CONNECTOR_CONFIG Config,
    [in] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out] UCMCONNECTOR        *Connector
);
```

## Parameters

[in] WdfDevice

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

[in] Config

A pointer to a caller-supplied [UCM\\_CONNECTOR\\_CONFIG](#) structure that is initialized by calling [UCM\\_CONNECTOR\\_CONFIG\\_INIT](#).

[in] Attributes

A pointer to a caller-supplied [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that contains attributes for the new connector object. This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

[out] Connector

A pointer to a location that receives a handle to the new connector object.

## Return value

**UcmConnectorCreate** returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

If the client driver specifies a connector identifier that is already in use, the method fails with STATUS\_INVALID\_PARAMETER error code.

If the Type-C connector is specified to be a Dual-Role port (DRP), the client driver must register its [EVT\\_UCM\\_CONNECTOR\\_SET\\_DATA\\_ROLE](#) event callback.

The parent object is WdfDevice. You can set the **ParentObject** member of [WDF\\_OBJECT\\_ATTRIBUTES](#) to NULL or the WDFDEVICE handle. The connector object gets deleted when the parent WDFDEVICE object gets deleted.

An appropriate place for a UCM client driver to call **UcmConnectorCreate** is in [EvtDevicePrepareHardware](#) or [EvtDeviceD0Entry](#). Conversely, the driver should release the UCMCONNECTOR handle in [EvtDeviceReleaseHardware](#) or [EvtDeviceD0Exit](#).

## Examples

This example code shows how to create a Type-C connector that is PD-capable.

```
UCMCONNECTOR Connector;

UCM_CONNECTOR_CONFIG_INIT(&connCfg, 0);

UCM_CONNECTOR_TYPE_C_CONFIG_INIT(
    &connCfg.TypeCConfig,
    UcmTypeCOperatingModeDrp,
    UcmTypeCCurrentDefaultUsb | UcmTypeCCurrent1500mA |
    UcmTypeCCurrent3000mA);

connCfg.EvtSetDataRole = EvtSetDataRole;

UCM_CONNECTOR_PD_CONFIG_INIT(&connCfg.PdConfig, UcmPowerRoleSink | UcmPowerRoleSource);

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attr, CONNECTOR_CONTEXT);

status = UcmConnectorCreate(Device, &connCfg, &attr, &Connector);
if (!NT_SUCCESS(status))
{
    TRACE_ERROR(
        "UcmConnectorCreate failed with %!STATUS!.",

```

```
    status);
    goto Exit;
}

TRACE_INFO("UcmConnectorCreate() succeeded.");
```

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UCM\\_CONNECTOR\\_CONFIG](#)

[UCM\\_CONNECTOR\\_CONFIG\\_INIT](#)

# UcmConnectorDataDirectionChanged function (ucmmanager.h)

Article02/22/2024

Notifies the USB connector manager framework extension (UcmCx) with the new data role of a change in data role.

## Syntax

C++

```
void UcmConnectorDataDirectionChanged(
    [in] UCMCONNECTOR  Connector,
    [in] BOOLEAN        Success,
    [in] UCM_DATA_ROLE CurrentDataRole
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] Success

Used to indicate failure of a data-role swap that was initiated by UcmCx using [EVT\\_UCM\\_CONNECTOR\\_SET\\_DATA\\_ROLE](#).

If TRUE, the operation was successful. FALSE, otherwise.

[in] CurrentDataRole

A [UCM\\_TYPEC\\_PARTNER](#) value that indicates the new data role.

## Return value

None

## Remarks

**UcmConnectorDataDirectionChanged** returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this inline function can return an appropriate **NTSTATUS** value.

If the connector partner is attached, UcmCx updates the data role of the partner depending on the *CurrentDataRole* value. For example, if the client driver changes the data role to **UcmTypeCPortStateUfp**, UcmCx updates the role of the connector partner to **UcmTypeCPortStateDfp**.

UcmCx can change the data role of a connector, and invokes **EVT\_UCM\_CONNECTOR\_SET\_DATA\_ROLE**. In response to that call, the client should perform the DR\_Swap operation, and indicate success/failure of the operation by calling **UcmConnectorDataDirectionChanged**.

Alternatively, the client driver might choose to perform a role-swap autonomously, or the partner might perform a role-swap. In either case, when the role-swap has completed, the driver must report the new role to UcmCx using **UcmConnectorDataDirectionChanged**.

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

- [UcmConnectorCreate](#)

# UcmConnectorPdConnectionStateChanged function (ucmmanager.h)

Article02/22/2024

Notifies the USB connector manager framework extension (UcmCx) with the connection capabilities of the currently negotiated PD contract (if any).

## Syntax

C++

```
NTSTATUS UcmConnectorPdConnectionStateChanged(
    [in] UCMCONNECTOR                           Connector,
    [in] PUCM_CONNECTOR_PD_CONN_STATE_CHANGED_PARAMS Params
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] Params

Pointer to a [UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#) structure contains driver-supplied state of the connector.

## Return value

[UcmConnectorPdConnectionStateChanged](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# UcmConnectorPdPartnerSourceCaps function (ucmmanager.h)

Article 10/21/2021

Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the partner connector.

## Syntax

C++

```
NTSTATUS UcmConnectorPdPartnerSourceCaps(
    [in] UCMCONNECTOR           Connector,
    UCM_PD_POWER_DATA_OBJECT [] Pdos,
    [in] UCHAR                  PdoCount
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

Pdos

A caller-allocated array of [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structures that describes the power source capabilities.

[in] PdoCount

Number of elements in the array specified by *Pdos[]*.

## Return value

[UcmConnectorPdPartnerSourceCaps](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

When using a Type-C connector for charging by using the power delivery (PD) mechanism, the local connector queries the partner connector for its supported power sourcing capabilities. That query is not required if the partner connector is the power source because in that case, the local connector cached the initial advertisement when the partner connector was attached. If the source capabilities changed, it sends an update to the local connector.

If the partner connector is the power sink, the local connector port must query for the latest capabilities.

## Examples

```
UCM_PD_POWER_DATA_OBJECT Pdos[1];

UCM_PD_POWER_DATA_OBJECT_INIT_FIXED(&Pdos[0]);

Pdos[0].FixedSupplyPdo.VoltageIn50mV = 100;           // 5V
Pdos[0].FixedSupplyPdo.MaximumCurrentIn10mA = 150;    // 1.5 A

status = UcmConnectorPdPartnerSourceCaps(
    Connector,
    Pdos,
    ARRSIZE(Pdos));
if (!NT_SUCCESS(status))
{
    TRACE_ERROR(
        "UcmConnectorPdPartnerSourceCaps() failed with %!STATUS! .",
        status);
    goto Exit;
}
```

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15

Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# UcmConnectorPdSourceCaps function (ucmmanager.h)

Article 10/21/2021

Notifies the USB connector manager framework extension (UcmCx) with the power source capabilities of the connector.

## Syntax

C++

```
NTSTATUS UcmConnectorPdSourceCaps(
    [in] UCMCONNECTOR                 Connector,
    UCM_PD_POWER_DATA_OBJECT [] Pdos,
    [in] UCHAR                         PdoCount
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

Pdos

A caller-allocated array of [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structures that describes the power source capabilities.

[in] PdoCount

Number of elements in the array specified by *Pdos[]*.

## Return value

[UcmConnectorPdSourceCaps](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

If the connector (local connector) is the power source, the client driver can report the capabilities and changes to those capabilities to UcmCx by using **UcmConnectorPdSourceCaps**. If connector is a the power sink, report the advertised capabilities received from partner by calling **UcmConnectorPdPartnerSourceCaps**. The client driver must call **UcmConnectorPdPartnerSourceCaps** each time the partner re-advertises its capabilities.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

# UcmConnectorPowerDirectionChanged function (ucmmanager.h)

Article02/22/2024

Notifies the USB connector manager framework extension (UcmCx) with the new power role of the partner connector.

## Syntax

C++

```
void UcmConnectorPowerDirectionChanged(
    [in] UCMCONNECTOR    Connector,
    [in] BOOLEAN         Success,
    [in] UCM_POWER_ROLE CurrentPowerRole
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] Success

Used to indicate failure of a power-role swap that was initiated by UcmCx using [EVT\\_UCM\\_CONNECTOR\\_SET\\_POWER\\_ROLE](#).

If TRUE, the operation was successful. FALSE, otherwise.

[in] CurrentPowerRole

One of the [UCM\\_POWER\\_ROLE](#)-typed flags that indicates the new data role.

## Return value

None

## Remarks

**UcmConnectorPowerDirectionChanged** returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this inline function can return an appropriate **NTSTATUS** value.

If the connector partner is attached, UcmCx updates the power role of the partner depending on the *CurrentPowerRole* value.

UcmCx can change the power role of a connector, and invokes **EVT\_UCM\_CONNECTOR\_SET\_POWER\_ROLE**. In response to that call, the client should perform the PR\_Swap operation, and indicate success/failure of the operation by calling **UcmConnectorPowerDirectionChanged**.

Alternatively, the client driver might choose to perform a role-swap autonomously, or the partner might perform a role-swap. In either case, when the role-swap has completed, the driver must report the new role to UcmCx using **UcmConnectorPowerDirectionChanged**.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

- [UcmConnectorCreate](#)

# UcmConnectorTypeCAttach function (ucmmanager.h)

Article 02/22/2024

Notifies the USB connector manager framework extension (UcmCx) when a partner connector is attached.

## Syntax

C++

```
NTSTATUS UcmConnectorTypeCAttach(
    [in] UCMCONNECTOR                 Connector,
    [in] PUCM_CONNECTOR_TYPEC_ATTACH_PARAMS Params
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] Params

A pointer to a driver-allocated [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#) that has been initialized by calling [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS\\_INIT](#).

## Return value

[UcmConnectorTypeCAttach](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

When a connection to a partner connector is detected, the client driver calls this method to notify UcmCx with information about the partner connector. That information includes the connector role, down stream or upstream facing port, the amount of current connector can draw or deliver, and charging state. UcmCx uses that information

to perform certain operations. For example, it may determine the role of the partner connector attached, and configure the USB controller in host or peripheral mode.

Typically, every **UcmConnectorTypeCAttach** call has a subsequent **UcmConnectorTypeCDetach** call to notify UcmCx when the partner connector is detached. However, when a powered cable without an upstream port is attached (indicated by **Params->PortPartnerType** set to **UcmTypeCPortStatePoweredCableNoUfp**). The client driver can call **UcmConnectorTypeCAttach** again when a connection is detected to the upstream port to the powered cable.

## Examples

```
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.");
```

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

<b>Requirement</b>	<b>Value</b>
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#)

[UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS\\_INIT](#)

[UcmConnectorCreate](#)

# UcmConnectorTypeCCurrentAdChanged function (ucmmanager.h)

Article 02/22/2024

Notifies the USB connector manager framework extension (UcmCx) when the specified connector changes the current advertisement. Either the connector changes it (when it is DFP/Source), or the partner changed it (when it is UFP/Sink).

## Syntax

C++

```
NTSTATUS UcmConnectorTypeCCurrentAdChanged(
    [in] UCMCONNECTOR      Connector,
    [in] UCM_TYPEC_CURRENT CurrentAdvertisement
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

[in] CurrentAdvertisement

The new current advertisement of the connector indicated by one of the [UCM\\_TYPEC\\_CURRENT](#)-typed flags.

## Return value

[UcmConnectorTypeCCurrentAdChanged](#) returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

When using a Type-C connector for charging, the partner connector sends a current advertisement when it's attached to the local connector. That initial advertisement is reported to UcmCx by calling [UcmConnectorTypeCAttach](#). During the lifetime of the

connection, the current level advertised by the source might change. The client driver must notify UcmCx about that change by calling method.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorCreate](#)

[UcmConnectorTypeCAttach](#)

[UcmConnectorTypeCCurrentAdChanged](#)

# UcmConnectorTypeCDetach function (ucmmanager.h)

Article02/22/2024

Notifies the USB connector manager framework extension (UcmCx) when the partner connector detaches from the specified Type-C connector.

## Syntax

C++

```
NTSTATUS UcmConnectorTypeCDetach(
    [in] UCMCONNECTOR Connector
);
```

## Parameters

[in] Connector

Handle to the connector object that the client driver received in the previous call to [UcmConnectorCreate](#).

## Return value

**UcmConnectorTypeCDetach** returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Requirement	Value
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmConnectorTypeCAttach](#)

# UcmInitializeDevice function (ucmmanager.h)

Article 02/22/2024

Initializes the USB connector manager framework extension (UcmCx).

## Syntax

C++

```
NTSTATUS UcmInitializeDevice(
    [in] WDFDEVICE           WdfDevice,
    [in] PUCM_MANAGER_CONFIG Config
);
```

## Parameters

[in] WdfDevice

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

[in] Config

A pointer to a caller-supplied [UCM\\_MANAGER\\_CONFIG](#) structure that is initialized by calling [UCM\\_MANAGER\\_CONFIG\\_INIT](#).

## Return value

`UcmInitializeDevice` returns `STATUS_SUCCESS` if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

This method initializes UcmCx and allocates resources required, registers for PnP events, and sets up I/O targets. The client driver must call this method in the driver's [EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#) implementation.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmmanager.h (include Ucmcx.h)
Library	UcmCxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UCM\\_MANAGER\\_CONFIG](#)

[UCM\\_MANAGER\\_CONFIG\\_INIT](#)

# ucmtcpcidevice.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucmtcpcidevice.h contains the following programming interfaces:

## Functions

### [UCMTCPCI\\_DEVICE\\_CONFIG\\_INIT](#)

Initializes the UCMTCPCI\_DEVICE\_CONFIG structure.

### [UcmTcpciDeviceInitialize](#)

Initializes the USB Type-C Port Controller Interface framework extension (UcmTcpciCx).

### [UcmTcpciDeviceInitInitialize](#)

Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

## Structures

### [UCMTCPCI\\_DEVICE\\_CONFIG](#)

Used in the client driver's call to UcmTcpciDeviceInitialize. Call UCMTCPCLI\_DEVICE\_CONFIG\_INIT to initialize this structure.

# UCMTCPCI\_DEVICE\_CONFIG structure (ucmtcpdevice.h)

Article02/22/2024

Used in the client driver's call to [UcmTcpciDeviceInitialize](#). Call [UCMTCPCI\\_DEVICE\\_CONFIG\\_INIT](#) to initialize this structure.

## Syntax

C++

```
typedef struct _UCMTCPCI_DEVICE_CONFIG {
    ULONG Size;
} UCMTCPCI_DEVICE_CONFIG, *PUCMTCPCI_DEVICE_CONFIG;
```

## Members

Size

Size of this structure.

## Requirements

[+] Expand table

Requirement	Value
Header	ucmtcpdevice.h

## See also

[UcmTcpciDeviceInitialize](#)

# UCMTCPCI\_DEVICE\_CONFIG\_INIT function (ucmtcpcidevice.h)

Article 02/22/2024

Initializes the [UCMTCPCI\\_DEVICE\\_CONFIG](#) structure.

## Syntax

C++

```
void UCMTCPCI_DEVICE_CONFIG_INIT(
    [out] PUCMTCPCI_DEVICE_CONFIG Config
);
```

## Parameters

[out] Config

A pointer to the driver-allocated [UCMTCPCI\\_DEVICE\\_CONFIG](#) structure.

## Return value

None

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpcidevice.h
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciDeviceInitialize](#)

# UcmTpcciDeviceInitialize function (ucmtpcidevice.h)

Article 02/22/2024

Initializes the USB Type-C Port Controller Interface framework extension (UcmTpcciCx).

## Syntax

C++

```
NTSTATUS UcmTpcciDeviceInitialize(
    WDFDEVICE                 WdfDevice,
    PUCMTCPCI_DEVICE_CONFIG Config
);
```

## Parameters

`WdfDevice`

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

`Config`

A pointer to a caller-supplied [UCMTCPCI\\_DEVICE\\_CONFIG](#) structure that is initialized by calling [UCMTCPCI\\_DEVICE\\_CONFIG\\_INIT](#). This value cannot be NULL.

## Return value

(NTSTATUS) The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method may return an appropriate [NTSTATUS](#) error code.

[+] Expand table

Return code	Description
<code>STATUS_INFO_LENGTH_MISMATCH</code>	Invalid size for the structure pointed to by <i>Config</i> . Must be size of <a href="#">UCMTCPCI_DEVICE_CONFIG</a> .
<code>STATUS_INVALID_DEVICE_STATE</code>	The Plug and Play state of the framework device object's is uninitialized. Call <a href="#">UcmTpcciDeviceInitialize</a> within the

driver's implementation of [EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#).

## Remarks

The client driver must call [UcmTcpciDeviceInitialize](#) within the driver's implementation of [EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#). This method configures the framework device object and allocates resources required, registers for PnP events, and sets up I/O targets.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpcidevice.h
Library	Ucmtcpcicxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[WdfDeviceCreate](#)

# UcmTcpciDeviceInitInitialize function (ucmtcpcidevice.h)

Article02/22/2024

Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

## Syntax

C++

```
NTSTATUS UcmTcpciDeviceInitInitialize(
    [in] PWDDEVICE_INIT DeviceInit
);
```

## Parameters

[in] DeviceInit

A pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure.

## Return value

(NTSTATUS) The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method may return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver calls this method after it has performed all of its own initialization in the [WDFDEVICE\\_INIT](#) structure, just before it calls [WdfDeviceCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1607

Requirement	Value
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpdevice.h
Library	Ucmtcpicxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[WDFDEVICE\\_INIT](#)

# ucmtcpciglobals.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

ucmtcpciglobals.h contains the following programming interfaces:

## Structures

### [UCMTCPCI\\_DRIVER\\_GLOBALS](#)

The global structure for the USB Type-C Port Controller Interface framework extension (UcmTcpciCx).

# UCMTCPCI\_DRIVER\_GLOBALS structure (ucmtcpciglobals.h)

Article02/22/2024

The global structure for the USB Type-C Port Controller Interface framework extension (UcmTcpciCx).

## Syntax

C++

```
typedef struct _UCMTCPCI_DRIVER_GLOBALS {
    ULONG Reserved;
} UCMTCPCI_DRIVER_GLOBALS, *PUCMTCPCI_DRIVER_GLOBALS;
```

## Members

### Reserved

Reserved.

## Requirements

  Expand table

Requirement	Value
Header	ucmtcpciglobals.h

# ucmtcpciportcontroller.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

ucmtcpciportcontroller.h contains the following programming interfaces:

## Functions

<a href="#">UCMTCPCI_PORT_CONTROLLER_ALERT_DATA_INIT</a>
Initializes the UCMTCPCI_PORT_CONTROLLER_ALERT_DATA structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_CAPABILITIES_INIT</a>
Initializes the UCMTCPCI_PORT_CONTROLLER_CAPABILITIES structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_CONFIG_INIT</a>
Initializes the UCMTCPCI_PORT_CONTROLLER_CONFIG structure.
<a href="#">UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION_INIT</a>
Initializes the UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION structure.
<a href="#">UcmTcpciPortControllerAlert</a>
Sends information about the hardware alerts that are received on the port controller to UcmTcpciCx.
<a href="#">UcmTcpciPortControllerCreate</a>
Creates a port controller object to register with UcmTcpciCx.
<a href="#">UcmTcpciPortControllerSetHardwareRequestQueue</a>
Assigns a framework queue object to which the UcmTcpciCx dispatches hardware requests for the port controller.
<a href="#">UcmTcpciPortControllerStart</a>
Indicates to the UcmTcpciCx class extension that the client driver is now ready to service hardware requests for the port controller.

## [UcmTcpciPortControllerStop](#)

Indicates to the UcmTcpciCx class extension to stop sending hardware requests to the port controller object.

# Structures

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_DATA](#)

Contains information about hardware alerts received on the port controller object. This structure is used in the UcmTcpciPortControllerAlert call. Call UCMTPCI\_PORT\_CONTROLLER\_ALERT\_DATA\_INIT to initialize this structure.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#)

Contains information about the capabilities of the port controller.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#)

Contains configuration options for the port controller object, passed by the client driver in the call to UcmTcpciPortControllerCreate. Call UCMTPCI\_PORT\_CONTROLLER\_CONFIG\_INIT to initialize this structure.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#)

Contains identification information and USB specification version information (in BCD format) about the port controller.

# Enumerations

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_TYPE](#)

Defines generic alert values that are used to indicate the type of hardware alert received on the port controller.

# UCMTCPCI\_PORT\_CONTROLLER\_ALERT\_DATA structure (ucmtcpciportcontroller.h)

Article02/22/2024

Contains information about hardware alerts received on the port controller object. This structure is used in the [UcmTcpciPortControllerAlert](#) call. Call [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_DATA\\_INIT](#) to initialize this structure.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_ALERT_DATA {
    ULONG             Size;
    UCMTCPCI_PORT_CONTROLLER_ALERT_TYPE AlertType;
    union {
        UCMTCPCI_PORT_CONTROLLER_CC_STATUS      CCStatus;
        UCMTCPCI_PORT_CONTROLLER_POWER_STATUS   PowerStatus;
        UCMTCPCI_PORT_CONTROLLER_FAULT_STATUS   FaultStatus;
        PUCMTCPCI_PORT_CONTROLLER_RECEIVE_BUFFER ReceiveBuffer;
    };
} UCMTCPCI_PORT_CONTROLLER_ALERT_DATA,
*PUCMTCPCI_PORT_CONTROLLER_ALERT_DATA;
```

## Members

Size

Size of this structure.

AlertType

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_TYPE](#) value that indicates the type of hardware alert.

CCStatus

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_CC\\_STATUS](#) structure that contains status information about the CC lines of the port controller. This structure is defined in [UcmTcpciSpec.h](#).

#### PowerStatus

A UCMTPCI\_PORT\_CONTROLLER\_POWER\_STATUS structure that contains the power status of the port controller. This structure is defined in UcmTcpciSpec.h.

#### FaultStatus

A UCMTPCI\_PORT\_CONTROLLER\_FAULT\_STATUS structure that contains the fault status of the port controller. This structure is defined in UcmTcpciSpec.h.

#### ReceiveBuffer

A pointer to a UCMTPCI\_PORT\_CONTROLLER\_RECEIVE\_BUFFER structure that represents the buffer for receiving the alert from the port controller. This structure is defined in UcmTcpciSpec.h.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucmtcpciportcontroller.h

## See also

[UcmTcpciPortControllerAlert](#)

# UCMTCPCI\_PORT\_CONTROLLER\_ALERT\_DATA\_INIT function (ucmtcpicportcontroller.h)

Article02/22/2024

Initializes the UCMTCPCI\_PORT\_CONTROLLER\_ALERT\_DATA structure.

Call this function before calling [UcmTcpciPortControllerAlert](#).

## Syntax

C++

```
void UCMTCPCI_PORT_CONTROLLER_ALERT_DATA_INIT(
    [out] PUCMTCPCI_PORT_CONTROLLER_ALERT_DATA AlertData
);
```

## Parameters

[out] AlertData

A pointer to the driver-allocated UCMTCPCI\_PORT\_CONTROLLER\_ALERT\_DATA structure.

## Return value

None

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Requirement	Value
Header	ucmtcpportcontroller.h
IRQL	PASSIVE_LEVEL

# UCMTCPCI\_PORT\_CONTROLLER\_ALERT\_TYPE enumeration (ucmtcpciportcontroller.h)

Article 06/03/2021

Defines generic alert values that are used to indicate the type of hardware alert received on the port controller.

## Syntax

C++

```
typedef enum _UCMTCPCI_PORT_CONTROLLER_ALERT_TYPE {
    UcmTcpciPortControllerAlertInvalid,
    UcmTcpciPortControllerAlertCCStatus,
    UcmTcpciPortControllerAlertPowerStatus,
    UcmTcpciPortControllerAlertReceiveSOPMessageStatus,
    UcmTcpciPortControllerAlertReceivedHardReset,
    UcmTcpciPortControllerAlertTransmitSOPMessageFailed,
    UcmTcpciPortControllerAlertTransmitSOPMessageDiscarded,
    UcmTcpciPortControllerAlertTransmitSOPMessageSuccessful,
    UcmTcpciPortControllerAlertVbusVoltageAlarmHi,
    UcmTcpciPortControllerAlertVbusVoltageAlarmLo,
    UcmTcpciPortControllerAlertFault,
    UcmTcpciPortControllerAlertRxBufferOverflow,
    UcmTcpciPortControllerAlertVbusSinkDisconnectDetected
} UCMTCPCI_PORT_CONTROLLER_ALERT_TYPE;
```

## Constants

`UcmTcpciPortControllerAlertInvalid`

The alert is invalid.

`UcmTcpciPortControllerAlertCCStatus`

Indicates a

CC status change alert.

`UcmTpcciPortControllerAlertPowerStatus`

Indicates a

power status change alert.

`UcmTpcciPortControllerAlertReceiveSOPMessageStatus`

Indicates an SOP message alert.

`UcmTpcciPortControllerAlertReceivedHardReset`

Indicates a hard Reset alert.

`UcmTpcciPortControllerAlertTransmitSOPMessageFailed`

Indicates that the SOP message transmission was not successful.

`UcmTpcciPortControllerAlertTransmitSOPMessageDiscarded`

Indicates that the

SOP message transmission was not sent due to an incoming receive message.

`UcmTpcciPortControllerAlertTransmitSOPMessageSuccessful`

Indicates that the SOP message transmission was successful.

`UcmTpcciPortControllerAlertVbusVoltageAlarmHi`

Indicates a high-voltage alarm.

`UcmTpcciPortControllerAlertVbusVoltageAlarmLo`

Indicates a low-voltage alarm.

`UcmTpcciPortControllerAlertFault`

Indicates that a Fault has occurred.

`UcmTpcciPortControllerAlertRxBufferOverflow`

Indicates that the

TCP/C Rx buffer has overflowed.

`UcmTpcciPortControllerAlertVbusSinkDisconnectDetected`

Indicates that a VBUS Sink Disconnect Threshold crossing has been detected

## Requirements

Header

ucmtpciportcontroller.h

## See also

[UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_DATA](#)

# UCMTCPCI\_PORT\_CONTROLLER\_CAPABILITIES structure (ucmtcpportcontroller.h)

Article02/22/2024

Contains information about the capabilities of the port controller. This client driver must specify that information in the call to [UcmTcpciPortControllerCreate](#) during initialization. Call [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES\\_INIT](#) to initialize this structure.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_CAPABILITIES {
    ULONG                                     Size;
    BOOLEAN
    IsPowerDeliveryCapable;
    UCMTCPCI_PORT_CONTROLLER_DEVICE_CAPABILITIES_1      DeviceCapabilities1;
    UCMTCPCI_PORT_CONTROLLER_DEVICE_CAPABILITIES_2      DeviceCapabilities2;
    UCMTCPCI_PORT_CONTROLLER_STANDARD_INPUT_CAPABILITIES
    StandardInputCapabilities;
    UCMTCPCI_PORT_CONTROLLER_STANDARD_OUTPUT_CAPABILITIES
    StandardOutputCapabilities;
} UCMTCPCI_PORT_CONTROLLER_CAPABILITIES,
*PUCMTCPCI_PORT_CONTROLLER_CAPABILITIES;
```

## Members

`Size`

The size of this structure.

`IsPowerDeliveryCapable`

Indicates whether the port controller supports [USB Power Delivery](#).

`DeviceCapabilities1`

A UCMTCPCI\_PORT\_CONTROLLER\_DEVICE\_CAPABILITIES\_1 structure that describes the DEVICE\_CAPABILITIES\_1 Register as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

## DeviceCapabilities2

A UCMTPCI\_PORT\_CONTROLLER\_DEVICE\_CAPABILITIES\_2 structure that describes the DEVICE\_CAPABILITIES\_2 Register as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTpCpISpec.h.

## StandardInputCapabilities

A UCMTPCI\_PORT\_CONTROLLER\_STANDARD\_INPUT\_CAPABILITIES structure that describes the STANDARD\_INPUT\_CAPABILITIES Register as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTpCpISpec.h.

## StandardOutputCapabilities

A UCMTPCI\_PORT\_CONTROLLER\_STANDARD\_OUTPUT\_CAPABILITIES structure that describes the STANDARD\_OUTPUT\_CAPABILITIES Register as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTpCpISpec.h.

# Requirements

[ ] Expand table

Requirement	Value
Header	ucmtpciportcontroller.h

## See also

[UCMTPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES\\_INIT](#)

[UcmTpCpIPortControllerCreate](#)

# UCMTCPCI\_PORT\_CONTROLLER\_CAPABILITIES\_INIT function (ucmtcpciportcontroller.h)

Article02/22/2024

Initializes the [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#) structure.

## Syntax

C++

```
void UCMTCPCI_PORT_CONTROLLER_CAPABILITIES_INIT(
    [out] PUCMTCPCI_PORT_CONTROLLER_CAPABILITIES Capabilities
);
```

## Parameters

[out] Capabilities

A pointer to the driver-allocated [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#) structure.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h

Requirement	Value
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciPortControllerCreate](#)

# UCMTCPCI\_PORT\_CONTROLLER\_CONFIG structure (ucmtcpciportcontroller.h)

Article 02/22/2024

Contains configuration options for the port controller object, passed by the client driver in the call to [UcmTcpciPortControllerCreate](#). Call [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG\\_INIT](#) to initialize this structure.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_CONFIG {
    ULONG Size;
    PUCMTCPCI_PORT_CONTROLLER_IDENTIFICATION Identification;
    PUCMTCPCI_PORT_CONTROLLER_CAPABILITIES Capabilities;
} UCMTCPCI_PORT_CONTROLLER_CONFIG, *PUCMTCPCI_PORT_CONTROLLER_CONFIG;
```

## Members

Size

Size of this structure.

Identification

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#) structure.

Capabilities

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#) structure.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontroller.h

## See also

[UcmTcpciPortControllerCreate](#)

# UCMTCPCI\_PORT\_CONTROLLER\_CONFIG\_INIT function (ucmtcpicportcontroller.h)

Article 02/22/2024

Initializes the [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#) structure.

## Syntax

C++

```
void UCMTCPCI_PORT_CONTROLLER_CONFIG_INIT(
    [out] PUCMTCPCI_PORT_CONTROLLER_CONFIG      Config,
    [in]  PUCMTCPCI_PORT_CONTROLLER_IDENTIFICATION Identification,
    [in]  PUCMTCPCI_PORT_CONTROLLER_CAPABILITIES Capabilities
);
```

## Parameters

[out] Config

A pointer to the driver-allocated [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#) structure.

[in] Identification

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#) structure.

[in] Capabilities

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_CAPABILITIES](#) structure.

## Return value

None

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpportcontroller.h
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciPortControllerCreate](#)

# UCMTCPCI\_PORT\_CONTROLLER\_IDENTIFICATION structure (ucmtcpportcontroller.h)

Article02/22/2024

Contains identification information and USB specification version information (in BCD format) about the port controller. This client driver must specify that information in the call to [UcmTcpciPortControllerCreate](#) during initialization. Call [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION\\_INIT](#) to initialize this structure.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION {
    ULONG  Size;
    UINT16 VendorId;
    UINT16 ProductId;
    UINT16 DeviceId;
    UINT16 TypeCRevisionInBcd;
    UINT16 PDRevisionAndVersionInBcd;
    UINT16 PDIInterfaceRevisionAndVersionInBcd;
} UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION,
*PUCMTCPCI_PORT_CONTROLLER_IDENTIFICATION;
```

## Members

Size

Size of this structure.

VendorId

Specifies the vendor identifier assigned by the USB specification committee.

ProductId

Specifies the product identifier. This value is assigned by the manufacturer.

DeviceId

The device ID for the USB Type-C port controller.

### TypeCRevisionInBcd

The revision ID for the USB Type-C port controller.

### PDRevisionAndVersionInBcd

The revision and version for the USB Type-C port controller that supports PD.

### PDIInterfaceRevisionAndVersionInBcd

The interface revision and version for the USB Type-C port controller that supports PD.

## Requirements

  [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontroller.h

## See also

[UcmTcpciPortControllerCreate](#)

# UCMTCPCI\_PORT\_CONTROLLER\_IDENTIFICATION\_INIT function (ucmtcpciportcontroller.h)

Article02/22/2024

Initializes the [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#) structure.

## Syntax

C++

```
void UCMTCPCI_PORT_CONTROLLER_IDENTIFICATION_INIT(
    [out] PUCMTCPCI_PORT_CONTROLLER_IDENTIFICATION Identification
);
```

## Parameters

[out] Identification

A pointer to the driver-allocated [UCMTCPCI\\_PORT\\_CONTROLLER\\_IDENTIFICATION](#) structure.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h

Requirement	Value
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciPortControllerCreate](#)

# UcmTcpciPortControllerAlert function (ucmtcpciportcontroller.h)

Article01/20/2022

Sends information about the hardware alerts that are received on the port controller to UcmTcpciCx.

## Syntax

C++

```
void UcmTcpciPortControllerAlert(
    [in] UCMTCPCI_PORTCONTROLLER          PortControllerObject,
    PUCMTCPCI_PORT_CONTROLLER_ALERT_DATA AlertData,
    size_t                                NumberOfAlerts
);
```

## Parameters

[in] PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

AlertData

A pointer to an array of [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALERT\\_DATA](#) that contains all current alerts that have not been sent to UcmTcpciCx. This value cannot be NULL.

NumberOfAlerts

The number of items in the array pointed to by *AlertData*. This value cannot be 0.

## Return value

None

## Remarks

**UcmTcpciPortControllerAlert** returns STATUS\_SUCCESS if the operation succeeds.

Otherwise, this inline function may return an appropriate [NTSTATUS](#) error code.

The client driver must call **UcmTcpciPortControllerAlert** that has been previously started by calling [UcmTcpciPortControllerStart](#).

When a hardware alert occurs, the client driver must determine the type of alerts, fetch any auxiliary information associated with that alert, such as a PD message, populate the array, and then call **UcmTcpciPortControllerAlert**.

The client driver must report the alerts sequentially. The driver must not call this method on threads that are running simultaneously as that can lead to race conditions. Even though the class extension ensures that all internal data is correctly lock-protected, if the driver calls **UcmTcpciPortControllerAlert** from multiple threads at the same time without any external synchronization, it is not guaranteed that set of received alerts is current. To avoid that scenario, the driver must call this method within the [Interrupt Service Routine](#) (ISR) or a [DPC object](#) that is queued for the ISR. The ISR should be synchronized correctly to have only one instance running at any given time.

The client driver must assume that the class extension may submit requests before **UcmTcpciPortControllerAlert** returns, from within this call.

When handling alerts, UcmTcpciCx may send hardware requests to the client driver before the **UcmTcpciPortControllerAlert** call returns. If the driver holds a lock while calling **UcmTcpciPortControllerAlert** and also attempts to acquire the same lock when handling the hardware request, deadlock can occur.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h
IRQL	<=DISPATCH_LEVEL

## See also

- [Handling Hardware Interrupts](#)



# UcmTcpciPortControllerCreate function (ucmtcpciportcontroller.h)

Article 02/22/2024

Creates a port controller object to register with UcmTcpciCx.

## Syntax

C++

```
NTSTATUS UcmTcpciPortControllerCreate(
    WDFDEVICE                               WdfDevice,
    PUCMTCPCI_PORT_CONTROLLER_CONFIG Config,
    PWDF_OBJECT_ATTRIBUTES      Attributes,
    UCMTCPCLIPORTCONTROLLER *PortControllerObject
);
```

## Parameters

`WdfDevice`

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

`Config`

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#) that is initialized by calling [UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG\\_INIT](#). This value cannot be NULL.

`Attributes`

A pointer to a [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that contains driver-supplied attributes for the new object. This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

`PortControllerObject`

A pointer to a location that receives a handle to the new port controller object.

## Return value

(NTSTATUS) The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method may return an appropriate [NTSTATUS](#) error code.

[+] [Expand table](#)

Return code	Description
STATUS_INVALID_DEVICE_REQUEST	The handle to a framework device object is invalid.
STATUS_INFO_LENGTH_MISMATCH	Invalid size for the structure pointed to by <i>Config</i> or <i>Config-&gt;Capabilities</i> . Must be size of <a href="#">UCMTCPCI_PORT_CONTROLLER_CONFIG</a> or <a href="#">UCMTCPCI_PORT_CONTROLLER_CAPABILITIES</a> , respectively.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h
Library	Ucmtcpcicxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#)

[UCMTCPCI\\_PORT\\_CONTROLLER\\_CONFIG](#)

[WdfDeviceCreate](#)

# **UcmTcpciPortControllerSetHardwareRequestQueue function (ucmtcpciportcontroller.h)**

Article02/22/2024

Assigns a framework queue object to which the UcmTcpciCx dispatches hardware requests for the port controller.

## Syntax

C++

```
void UcmTcpciPortControllerSetHardwareRequestQueue(
    UCMTCPICPORTCONTROLLER PortControllerObject,
    WDFQUEUE             HardwareRequestQueue
);
```

## Parameters

**PortControllerObject**

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

**HardwareRequestQueue**

A handle to the framework queue object to assign.

## Return value

None

## Remarks

The client driver must call [UcmTcpciPortControllerSetHardwareRequestQueue](#) after creating the port controller object. The driver must call this method only once before calling [UcmTcpciPortControllerStart](#).

The parent of the queue object is the port controller object.

A client driver may choose to use the same queue across multiple port controller objects. However, in that case the driver must make sure that the port controller objects do not outlive the queue object. The queue object must be deleted only after all the port controllers have been stopped. UcmTcpciCx guarantees that only one request is processed in the queue at a time per port controller object.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h
IRQL	<=DISPATCH_LEVEL

## See also

[UcmTcpciPortControllerCreate](#)

# UcmTcpciPortControllerStart function (ucmtcpciportcontroller.h)

Article02/22/2024

Indicates to the UcmTcpciCx class extension that the client driver is now ready to service hardware requests for the port controller.

## Syntax

C++

```
NTSTATUS UcmTcpciPortControllerStart(
    UCMTCPCHIPORTCONTROLLER PortControllerObject
);
```

## Parameters

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

## Return value

(NTSTATUS) The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method may return an appropriate [NTSTATUS](#) error code.

[+] Expand table

Return code	Description
STATUS_INVALID_DEVICE_REQUEST	The port controller is already in Start state.
STATUS_INVALID_HANDLE	Hardware request queue has not been set by calling <a href="#">UcmTcpciPortControllerSetHardwareRequestQueue</a> .

## Remarks

After the client driver has received the UCMPORTCONTROLLER handle for the port controller object, the driver calls this method to notify the class extension that the driver can start receiving hardware requests. This method call allows the client driver to perform initialization of its framework context space on the port controller object, before the class extension can invoke the driver's callback functions or requests for the port controller object. The driver cannot call [UcmTcpciPortControllerAlert](#) or [UcmTcpciPortControllerStop](#) until the port controller has been started.

The client driver calls this method right after calling [UcmTcpciPortControllerCreate](#) and initializing its context structure, if it was specified in the [WDF\\_OBJECT\\_ATTRIBUTES](#) structure as the *Attributes* parameter value. The driver must assume that the class extension may submit requests even before [UcmTcpciPortControllerStart](#) returns, i.e., from within this DDI call. If the driver is holding a lock while calling [UcmTcpciPortControllerStart](#) and also attempts to acquire a lock while handling a hardware request (in its hardware request queue callback), it might result in a deadlock.

A call to [UcmTcpciPortControllerStart](#) to start a port controller object already in Start state, results in an error.

On boot, if the BIOS had already negotiated a PD contract, UcmTcpciCx starts from an unattached state.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h
Library	UcmTcpcicxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciPortControllerStop](#)

# UcmTcpciPortControllerStop function (ucmtcpciportcontroller.h)

Article02/22/2024

Indicates to the UcmTcpciCx class extension to stop sending hardware requests to the port controller object.

## Syntax

C++

```
void UcmTcpciPortControllerStop(
    UCMTCPCHIPORTCONTROLLER PortControllerObject
);
```

## Parameters

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

## Return value

None

## Remarks

After calling [UcmTcpciPortControllerStop](#), the client driver stops processing all requests on the port controller object. This call is synchronous, so it is guaranteed that the class extension will not invoke callback functions or send requests after it returns. The driver must not call this method within a port controller callback, or while any non-cancelable hardware requests are pending.

A client driver calls this method from its [EVT\\_WDF\\_DEVICE\\_RELEASE\\_HARDWARE](#) callback implementation. After doing so, it should also call [WdfObjectDelete](#), in case [EVT\\_WDF\\_DEVICE\\_RELEASE\\_HARDWARE](#) is invoked to resource rebalancing. Failure to do so causes the driver to leak objects associated with the port controller object when a

resource rebalance occurs. Parenting the UCMPORTCONTROLLER handle to the WDFDEVICE handle is not sufficient, because a WDFDEVICE is not deleted across a resource rebalance.

If the driver is transitioning to a Dx state due to S0-IDLE, the driver must not call this method from its [EVT\\_WDF\\_DEVICE\\_D0\\_EXIT](#) callback function. Synchronization with the driver's power state can be achieved by using a power-managed queue to receive hardware requests.

It is safe to call [UcmTcpciPortControllerStop](#) on a port controller that has already been stopped. After this method returns, no other method except for [UcmTcpciPortControllerStart](#) can be called on the port controller.

The client driver must call this method if it needs to stop all actions on the port controller so that it can perform error recovery if it detected any issues during its operation. After the recovery process has been completed, the driver must restart the port controller.

Stopping the controller ends any active PD contract and the Type-C connection.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Header	ucmtcpciportcontroller.h
Library	Ucmtcpcicxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UcmTcpciPortControllerStart](#)

# ucmtcpciportcontrollerrequests.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

ucmtcpciportcontrollerrequests.h contains the following programming interfaces:

## IOCTLs

### [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.

### [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\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED](#)

Notifies the client driver that the display out status of the DisplayPort connection has changed 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.

### [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\\_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\\_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\\_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\\_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\\_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\\_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\\_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.

## Structures

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED\\_IN\\_PARAMS](#)

Stores information about the alternate mode that was detected. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_ENTERED request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED\\_IN\\_PARAMS](#)

Stores information about the alternate mode that was exited. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_EXITED request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED\\_IN\\_PARAMS](#)

Stores information about the pin assignment of the DisplayPort alternate mode that was configured. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_CONFIGURED request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#)

Stores information about display out status of the DisplayPort connection. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_DISPLAY\_OUT\_STATUS\_CHANGED request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#)

Stores information about hot plug detect status of the DisplayPort connection. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_HPD\_STATUS\_CHANGED request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_IN\\_PARAMS](#)

This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_CONTROL request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_OUT\\_PARAMS](#)

Stores the values of all control registers of the port controller retrieved by the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_CONTROL request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_IN\\_PARAMS](#)

This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_STATUS request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_OUT\\_PARAMS](#)

Stores the values of all status registers of the port controller. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_STATUS request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND\\_IN\\_PARAMS](#)

Stores the specified command registers. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_COMMAND request.

## [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT\\_IN\\_PARAMS](#)

Stores the value of the CONFIG\_STANDARD\_OUTPUT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONFIG\_STANDARD\_OUTPUT request.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL\\_IN\\_PARAMS](#)

Stores the values of all control registers. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONTROL request.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO\\_IN\\_PARAMS](#)

Stores the value of the VBUS\_VOLTAGE\_ALARM\_LO\_CFG Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_MESSAGE\_HEADER\_INFO request.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT\\_IN\\_PARAMS](#)

Stores the value of the RECEIVE\_DETECT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_RECEIVE\_DETECT request.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER\\_IN\\_PARAMS](#)

Stores the value of the TRANSMIT\_BUFFER Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT\_BUFFER request.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_IN\\_PARAMS](#)

Stores the values of TRANSMIT Register. This structure is used in the IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT request.

## Enumerations

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS](#)

Defines values to determine whether a display out status for a DisplayPort device is enabled.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS](#)

Defines values to determine whether a DisplayPort device is plugged in.

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#)

Learn more about: [\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#) enumeration

#### [UCMTCPCI\\_PORT\\_CONTROLLER\\_IOCTL](#)

Defines the various device I/O control requests that are sent to the client driver for the port controller. This indicates the type of IOCTL in WPP.

# IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_ENTERED IOCTL (ucmtcpicportcontrollerrequests.h)

Article02/22/2024

Notifies the client driver that an alternate mode is entered so that the driver can perform additional tasks.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED\\_IN\\_PARAMS](#) structure that contains information about the alternate mode.

## Input buffer length

The size of [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED\\_IN\\_PARAMS](#).

## Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

The UcmTcpciCx class extension sends this IOCTL request when an alternate mode is entered. The client driver can determine the alternate mode that was entered based on the values passed in `SVID` and `Mode` of the supplied structure.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_EXITED IOCTL (ucmtcpicportcontrollerrequests.h)

Article02/22/2024

Notifies the client driver that an alternate mode is exited so that the driver can perform additional tasks.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED\\_IN\\_PARAMS](#) structure that contains information about the alternate mode.

## Input buffer length

The size of [UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED\\_IN\\_PARAMS](#).

## Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

The UcmTcpcCx class extension sends this IOCTL request when an alternate mode is exited. The client driver can determine the alternate mode that was entered based on the values passed in `SVID` and `Mode` of the supplied structure.

## Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_CONFIGURED IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

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.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED\\_IN\\_PARAMS](#) structure that contains information about the pin assignment.

## **Input buffer length**

The size of [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED\\_IN\\_PARAMS](#).

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request when the DisplayPort mode is configured. The client driver can determine the pin assignment based on the values passed in the supplied structure.

## **Requirements**

[\[\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_DISPLAY\_OUT\_STATUS\_CHANGED IOCTL**

## **(ucmtcpciportcontrollerrequests.h)**

Article02/22/2024

Notifies the client driver that the display out status of the DisplayPort connection has changed so that the driver can perform additional tasks.

### **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#) structure that contains status information.

### **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#) structure.

### **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

### **Remarks**

The UcmTcpciCx class extension sends this IOCTL request when the display out status changes. The client driver can determine the new status based on the values passed in the supplied structure.

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucmtcpciportcontrollerrequests.h
IRQL	PASSIVE_LEVEL

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_HPD\_STATUS\_CHANGED IOCTL**

## **(ucmtcpciportcontrollerrequests.h)**

Article 02/22/2024

Notifies the client driver that the hot-plug detect status of the DisplayPort connection has changed so that the driver can perform additional tasks.

### **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#) structure that contains status information.

### **Input buffer length**

The size of [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#).

### **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

### **Remarks**

The UcmTcpciCx class extension sends this IOCTL request when the DisplayPort mode status changes. The client driver can determine the new status based on the values passed in the supplied structure.

### **Requirements**

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ GET\_CONTROL IOCTL (ucmtcpicportcontrollerrequests.h)**

Article05/18/2021

Gets the values of all control registers defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_IN\\_PARAMS](#) structure that contains all control register values. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_IN\\_PARAMS](#) structure.

## **Output buffer**

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_OUT\\_PARAMS](#) structure. To get the structure, call [WdfRequestRetrieveOutputBuffer](#) by passing the received framework request object.

## **Output buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_OUT\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

The UcmTcpciCx class extension sends this IOCTL request to retrieve the values of the control registers. The client driver must communicate with the port controller to retrieve the POWER\_CONTROL, ROLE\_CONTROL, TCPC\_CONTROL, and FAULT\_CONTROL Register values and populate the received

[UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL\\_OUT\\_PARAMS](#) structure with those values. To complete the request, the driver must set the populated structure on the framework request object by calling [WdfRequestSetInformation](#) and then call [WdfRequestComplete](#) to complete the request.

## Requirements

Header	ucmtcpciportcontrollerrequests.h
--------	----------------------------------

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_ GET\_STATUS IOCTL (ucmtcpicportcontrollerrequests.h)**

Article05/18/2021

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.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_IN\\_PARAMS](#) structure that contains all control register values. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_IN\\_PARAMS](#) structure.

## **Output buffer**

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_OUT\\_PARAMS](#) structure. To get the structure, call [WdfRequestRetrieveOutputBuffer](#) by passing the received framework request object.

## **Output buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_OUT\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

The UcmTcpciCx class extension sends this IOCTL request to retrieve the values of the status registers. The client driver must communicate with the port controller to retrieve the register values and populate the received [UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS\\_OUT\\_PARAMS](#) structure with those values. To complete the request, the driver must set the populated structure on the framework request object by calling [WdfRequestSetInformation](#) and then call [WdfRequestComplete](#) to complete the request.

## Requirements

Header	ucmtcpciportcontrollerrequests.h
--------	----------------------------------

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_COMMAND IOCTL (ucmtcpicportcontrollerrequests.h)**

Article 02/22/2024

Sets the value of a command register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND\\_IN\\_PARAMS](#) structure. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the value of the command register. The value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONFIG\_STANDARD\_OUTPUT IOCTL**

## **(ucmtcpciportcontrollerrequests.h)**

Article 02/22/2024

Sets the CONFIG\_STANDARD\_OUTPUT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

### **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### **Input buffer**

A pointer to a

[UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT\\_IN\\_PARAMS](#) structure that contains the values to set in the CONFIG\_STANDARD\_OUTPUT Register. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object. This structure is declared in UcmTcpciSpec.h.

### **Input buffer length**

The size of the

[UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT\\_IN\\_PARAMS](#) structure.

### **Status block**

**Irp->IoStatus.Status** is set to STATUS\_SUCCESS if the request is successful. Otherwise, Status to the appropriate error condition as a [NTSTATUS](#) code.

### **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the CONFIG\_STANDARD\_OUTPUT Register. The value to set is provided in the supplied

structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## Requirements

[+] Expand table

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONTROL IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

Sets the value of a control register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL\\_IN\\_PARAMS](#) structure that contains the type of register and the value to set.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set values to the control register. Only one register can be set per request. The type and value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

 Expand table

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_MESSAGE\_HEADER\_INFO IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

Sets the value of the MESSAGE\_HEADER\_INFO Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a

[UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO\\_IN\\_PARAMS](#) structure. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object.

## **Input buffer length**

The size of the

[UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the value of the `VBUS_VOLTAGE_ALARM_LO_CFG` Register. The value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_RECEIVE\_DETECT IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

Sets the RECEIVE\_DETECT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT\\_IN\\_PARAMS](#) structure that contains the value to set. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the RECEIVE\_DETECT Register. The value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

Sets the TRANSMIT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_IN\\_PARAMS](#) structure that contains the value to set in the TRANSMIT Register. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object. This structure is declared in UcmTcpciSpec.h.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the TRANSMIT Register. The value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT\_BUFFER IOCTL (ucmtcpicportcontrollerrequests.h)**

Article02/22/2024

Sets the TRANSMIT\_BUFER Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER\\_IN\\_PARAMS](#) structure that contains the value to set in the TRANSMIT\_BUFFER Register. To get the structure, call [WdfRequestRetrieveInputBuffer](#) by passing the received framework request object. This structure is declared in UcmTcpciSpec.h.

## **Input buffer length**

The size of the [UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER\\_IN\\_PARAMS](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` to the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

The UcmTcpciCx class extension sends this IOCTL request to set the TRANSMIT\_BUFFER Register. The value to set is provided in the supplied structure. After setting the value in the register, client driver must call [WdfRequestComplete](#) to complete the request.

## **Requirements**

Requirement	Value
Header	ucmtcpportcontrollerrequests.h

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_ENTERED\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores information about the alternate mode that was detected. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_ENTERED_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER PortControllerObject;
    UINT16                  SVID;
    UINT32                  Mode;
} UCMTCPCHIPORTCONTROLLER_ALTERNATE_MODE_ENTERED_IN_PARAMS,
*PUCMTCPCHIPORTCONTROLLER_ALTERNATE_MODE_ENTERED_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

SVID

The Standard or Vendor ID (SVID) for the alternate mode that was entered. In Windows 10, version 1703, the supported value is DISPLAYPORT\_SVID, indicating that the partner device has entered DisplayPort mode.

Mode

The Standard or Vendor defined Mode value for the alternate mode that was entered.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED](#)

# UCMTCPCI\_PORT\_CONTROLLER\_ALTERNATE\_MODE\_EXITED\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores information about the alternate mode that was exited. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_EXITED_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER PortControllerObject;
    UINT16                  SVID;
    UINT32                  Mode;
} UCMTCPCHIPORTCONTROLLER_ALTERNATE_MODE_EXITED_IN_PARAMS,
*PUCMTCPCHIPORTCONTROLLER_ALTERNATE_MODE_EXITED_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

SVID

The Standard or Vendor ID (SVID) for the alternate mode that was exited. In Windows 10, version 1703, the supported value is DISPLAYPORT\_SVID, indicating that the partner device has exited DisplayPort mode.

Mode

The Standard or Vendor defined Mode value for the alternate mode that was exited.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED](#)

# UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_CONFIGURED\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores information about the pin assignment of the DisplayPort alternate mode that was configured. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER                         PortControllerObject;
    UCMTCPICI_PORT_CONTROLLER_DISPLAYPORT_PIN_ASSIGNMENT PinAssignment;
} UCMTCPICI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED_IN_PARAMS,
*PUCMTCPICI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

PinAssignment

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#)-type value that indicates the pin that was configured.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED](#)

# UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_DISPLAY\_OUT\_STATUS enumeration (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Defines values to determine whether a display out status for a DisplayPort device is enabled

## Syntax

C++

```
typedef enum _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS {
    UcmTcpciPortControllerDisplayOutStatusOff,
    UcmTcpciPortControllerDisplayOutStatusOn
} UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS;
```

## Constants

[+] Expand table

<code>UcmTcpciPortControllerDisplayOutStatusOff</code> Display out status is enabled.
<code>UcmTcpciPortControllerDisplayOutStatusOn</code> Display out status is enabled.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED](#)

[UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED\\_IN\\_PARAMS](#)

# UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_DISPLAY\_OUT\_STATUS\_CHANGED\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores information about display out status of the DisplayPort connection. This structure is used in the

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED](#) request.

## Syntax

C++

```
typedef struct
    _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS_CHANGED_IN_PARAMS {
    UCMTCPPIPORTCONTROLLER
    PortControllerObject;
    UCMTCPPI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS DisplayOutStatus;
    UCMTCPPI_PORT_CONTROLLER_DISPLAYPORT_PIN_ASSIGNMENT PinAssignment;
} UCMTCPPI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS_CHANGED_IN_PARAMS,
*PUCMTCPPI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS_CHANGED_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

DisplayOutStatus

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS](#)-type value that indicates status.

PinAssignment

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_PIN\\_ASSIGNMENT](#)-type value that indicates the pin that was changed.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_DISPLAY\\_OUT\\_STATUS\\_CHANGED](#)

# UCMTCPCI\_PORT\_CONTROLLER\_DISPLA YPORT\_HPD\_STATUS enumeration (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Defines values to determine whether a DisplayPort device is plugged in.

## Syntax

C++

```
typedef enum _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS {  
    UcmTcpciPortControllerHPDStatusLow,  
    UcmTcpciPortControllerHPDStatusHigh  
} UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS;
```

## Constants

[ ] Expand table

<code>UcmTcpciPortControllerHPDStatusLow</code>	The DisplayPort device is unplugged.
<code>UcmTcpciPortControllerHPDStatusHigh</code>	A DisplayPort device such as a monitor is plugged in.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_HPD\_STATUS\_CHANGED

# UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_HPD\_STATUS\_CHANGED\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores information about hot plug detect status of the DisplayPort connection. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED_IN_PARAMS {
    UCMTCPICIOPORTCONTROLLER PortControllerObject;
    UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS HPDStatus;
} UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

HPDStatus

A [UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS](#)-type value that indicates status.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED](#)

# **UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_PIN\_ASSIGNMENT enumeration (ucmtcpciportcontrollerrequests.h)**

Article09/16/2021

The \_UCMTCPCI\_PORT\_CONTROLLER\_DISPLAYPORT\_PIN\_ASSIGNMENT enumeration defines values used to identify DisplayPort pin assignments.

## Syntax

C++

```
typedef enum _UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_PIN_ASSIGNMENT {
    UcmTcpciPortControllerPinAssignmentInvalid,
    UcmTcpciPortControllerDFPDPinAssignmentA,
    UcmTcpciPortControllerDFPDPinAssignmentB,
    UcmTcpciPortControllerDFPDPinAssignmentC,
    UcmTcpciPortControllerDFPDPinAssignmentD,
    UcmTcpciPortControllerDFPDPinAssignmentE,
    UcmTcpciPortControllerDFPDPinAssignmentF,
    UcmTcpciPortControllerUFPDPinAssignmentA,
    UcmTcpciPortControllerUFPDPinAssignmentB,
    UcmTcpciPortControllerUFPDPinAssignmentC,
    UcmTcpciPortControllerUFPDPinAssignmentD,
    UcmTcpciPortControllerUFPDPinAssignmentE
} UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_PIN_ASSIGNMENT;
```

## Constants

**UcmTcpciPortControllerPinAssignmentInvalid**

Invalid pin assignment

**UcmTcpciPortControllerDFPDPinAssignmentA**

DFPD pin assignment A

**UcmTcpciPortControllerDFPDPinAssignmentB**

DFPD pin assignment B

**UcmTcpciPortControllerDFPDPinAssignmentC**

DFPD pin assignment C

`UcmTcpciPortControllerDFPDPinAssignmentD`

DFPD pin assignment D

`UcmTcpciPortControllerDFPDPinAssignmentE`

DFPD pin assignment E

`UcmTcpciPortControllerDFPDPinAssignmentF`

DFPD pin assignment F

`UcmTcpciPortControllerUFPDPinAssignmentA`

UFPD pin assignment A

`UcmTcpciPortControllerUFPDPinAssignmentB`

UFPD pin assignment B

`UcmTcpciPortControllerUFPDPinAssignmentC`

UFPD pin assignment C

`UcmTcpciPortControllerUFPDPinAssignmentD`

UFPD pin assignment D

`UcmTcpciPortControllerUFPDPinAssignmentE`

UFPD pin assignment E

## Requirements

**Header**

`ucmtcpciportcontrollerrequests.h`

# UCMTCPCI\_PORT\_CONTROLLER\_GET\_CONTROL\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article02/22/2024

This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_GET_CONTROL_IN_PARAMS {
    UCMTCPCIPORTCONTROLLER PortControllerObject;
} UCMTCPCI_PORT_CONTROLLER_GET_CONTROL_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_GET_CONTROL_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL](#)

# UCMTCPCI\_PORT\_CONTROLLER\_GET\_CONTROL\_OUT\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores the values of all control registers of the port controller retrieved by the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_GET_CONTROL_OUT_PARAMS {
    UCMTCPCI_PORT_CONTROLLER_TCPC_CONTROL    TCPCControl;
    UCMTCPCI_PORT_CONTROLLER_ROLE_CONTROL    RoleControl;
    UCMTCPCI_PORT_CONTROLLER_FAULT_CONTROL   FaultControl;
    UCMTCPCI_PORT_CONTROLLER_POWER_CONTROL   PowerControl;
} UCMTCPCI_PORT_CONTROLLER_GET_CONTROL_OUT_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_GET_CONTROL_OUT_PARAMS;
```

## Members

TCPCControl

A UCMTCPCI\_PORT\_CONTROLLER\_TCPC\_CONTROL structure that describes the TCPC\_CONTROL Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

RoleControl

A UCMTCPCI\_PORT\_CONTROLLER\_ROLE\_CONTROL structure that describes the ROLE\_CONTROL Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

FaultControl

A UCMTCPCI\_PORT\_CONTROLLER\_FAULT\_CONTROL structure that describes the FAULT\_CONTROL Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

PowerControl

A UCMTPCI\_PORT\_CONTROLLER\_POWER\_CONTROL structure that describes the FAULT\_POWER Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTpCpISpec.h.

## Requirements

 Expand table

Requirement	Value
Header	ucmtpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL](#)

# UCMTCPCI\_PORT\_CONTROLLER\_GET\_STATUS\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article02/22/2024

This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_GET_STATUS_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER PortControllerObject;
} UCMTCPCHIPORTCONTROLLER_GET_STATUS_IN_PARAMS,
*PUCMTCPCHIPORTCONTROLLER_GET_STATUS_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

# UCMTCPCI\_PORT\_CONTROLLER\_GET\_STATUS\_OUT\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores the values of all status registers of the port controller. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_GET_STATUS_OUT_PARAMS {
    UCMTCPCI_PORT_CONTROLLER_CC_STATUS    CCStatus;
    UCMTCPCI_PORT_CONTROLLER_POWER_STATUS PowerStatus;
    UCMTCPCI_PORT_CONTROLLER_FAULT_STATUS FaultStatus;
} UCMTCPCI_PORT_CONTROLLER_GET_STATUS_OUT_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_GET_STATUS_OUT_PARAMS;
```

## Members

CCStatus

A UCMTCPCI\_PORT\_CONTROLLER\_CC\_STATUS structure that describes the CC\_STATUS Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

PowerStatus

A UCMTCPCI\_PORT\_CONTROLLER\_POWER\_STATUS structure that describes the POWER\_STATUS Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

FaultStatus

A UCMTCPCI\_PORT\_CONTROLLER\_FAULT\_STATUS structure that describes the FAULT\_STATUS Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

# Requirements

 Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS](#)

# UCMTCPCI\_PORT\_CONTROLLER\_IOCTL enumeration (ucmtcpicportcontrollerrequests.h)

Article 06/03/2021

Defines the various device I/O control requests that are sent to the client driver for the port controller. This indicates the type of IOCTL in WPP.

## Syntax

C++

```
typedef enum _UCMTCPCI_PORT_CONTROLLER_IOCTL {
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_GET_STATUS,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_GET_CONTROL,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_CONTROL,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_COMMAND,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_ENTERED,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_EXITED,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED,
    _IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS_CHANGED
} UCMTCPCI_PORT_CONTROLLER_IOCTL;
```

## Constants

\_IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_STATUS

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_STATUS](#) request.

\_IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_GET\_CONTROL

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_GET\\_CONTROL](#) request.

\_IOCTL\_UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONTROL

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_COMMAND`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_ENTERED`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_ENTERED](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_ALTERNATE_MODE_EXITED`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_ALTERNATE\\_MODE\\_EXITED](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_CONFIGURED`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_CONFIGURED](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_HPD_STATUS_CHANGED`

The [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_DISPLAYPORT\\_HPD\\_STATUS\\_CHANGED](#) request.

`_IOCTL_UCMTCPCI_PORT_CONTROLLER_DISPLAYPORT_DISPLAY_OUT_STATUS_CHANGED`

## Requirements

Header

ucmtcpciportcontrollerrequests.h

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_COMMAND\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article02/22/2024

Stores the specified command registers. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_COMMAND_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER          PortControllerObject;
    UCMTCPCI_PORT_CONTROLLER_COMMAND Command;
} UCMTCPCI_PORT_CONTROLLER_SET_COMMAND_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_SET_COMMAND_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

`Command`

A UCMTCPCI\_PORT\_CONTROLLER\_COMMAND-value that indicates the type of control register. This enumeration is declared in UcmTcpciSpec.h.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_COMMAND](#)

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONFIG\_STANDARD\_OUTPUT\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores the value of the CONFIG\_STANDARD\_OUTPUT Register. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT_IN_PARAMS {
    UCMTCPCIPORTCONTROLLER PortControllerObject;
    UCMTCPCI_PORT_CONTROLLER_CONFIG_STANDARD_OUTPUT ConfigStandardOutput;
} UCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_SET_CONFIG_STANDARD_OUTPUT_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

ConfigStandardOutput

A UCMTCPCI\_PORT\_CONTROLLER\_CONFIG\_STANDARD\_OUTPUT structure that describes the CONFIG\_STANDARD\_OUTPUT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in [UcmTcpciSpec.h](#).

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONFIG\\_STANDARD\\_OUTPUT](#)

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_CONTROL\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article04/01/2021

Stores the values of all control registers. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_CONTROL_IN_PARAMS {
    UCMTCPCIPORTCONTROLLER           PortControllerObject;
    UCMTCPCI_PORT_CONTROLLER_CONTROL_TYPE ControlType;
    union {
        UCMTCPCI_PORT_CONTROLLER_TCPC_CONTROL  TCPCCControl;
        UCMTCPCI_PORT_CONTROLLER_ROLE_CONTROL  RoleControl;
        UCMTCPCI_PORT_CONTROLLER_FAULT_CONTROL FaultControl;
        UCMTCPCI_PORT_CONTROLLER_POWER_CONTROL PowerControl;
    };
} UCMTCPCI_PORT_CONTROLLER_SET_CONTROL_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_SET_CONTROL_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

`ControlType`

A `UCMTCPCI_PORT_CONTROLLER_CONTROL_TYPE`-value that indicates the type of control register. This enumeration is declared in `UcmTcpciSpec.h`.

`TCPCCControl`

A `UCMTCPCI_PORT_CONTROLLER_TCPC_CONTROL` structure that describes the `TCPC_CONTROL` Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in `UcmTcpciSpec.h`.

#### **RoleControl**

A UCMTPCI\_PORT\_CONTROLLER\_ROLE\_CONTROL structure that describes the ROLE\_CONTROL Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

#### **FaultControl**

A UCMTPCI\_PORT\_CONTROLLER\_FAULT\_CONTROL structure that describes the FAULT\_CONTROL Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

#### **PowerControl**

A UCMTPCI\_PORT\_CONTROLLER\_POWER\_CONTROL structure that describes the FAULT\_POWER Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

## **Requirements**

Header	ucmtcpciportcontrollerrequests.h
--------	----------------------------------

## **See also**

[IOCTL\\_UCMTPCI\\_PORT\\_CONTROLLER\\_SET\\_CONTROL](#)

# **UCMTCPCI\_PORT\_CONTROLLER\_SET\_MESSAGE\_HEADER\_INFO\_IN\_PARAMS**

## **structure**

### **(ucmtcpciportcontrollerrequests.h)**

Article 02/22/2024

Stores the value of the VBUS\_VOLTAGE\_ALARM\_LO\_CFG Register. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER                    PortControllerObject;
    UCMTCPICI_PORT_CONTROLLER_MESSAGE_HEADER_INFO MessageHeaderInfo;
} UCMTCPICI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO_IN_PARAMS,
*PUCMTCPICI_PORT_CONTROLLER_SET_MESSAGE_HEADER_INFO_IN_PARAMS;
```

## Members

**PortControllerObject**

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

**MessageHeaderInfo**

A **UCMTCPCI\_PORT\_CONTROLLER\_MESSAGE\_HEADER\_INFO** structure that describes the MESSAGE\_HEADER\_INFO Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_MESSAGE\\_HEADER\\_INFO](#)

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_RECEIVE\_DETECT\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article02/22/2024

Stores the value of the RECEIVE\_DETECT Register. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_RECEIVE\\_DETECT](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER           PortControllerObject;
    UCMTCPCI_PORT_CONTROLLER_RECEIVE_DETECT ReceiveDetect;
} UCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT_IN_PARAMS,
*PUCMTCPCI_PORT_CONTROLLER_SET_RECEIVE_DETECT_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

`ReceiveDetect`

A UCMTCPCI\_PORT\_CONTROLLER\_RECEIVE\_DETECT structure that describes the RECEIVE\_DETECT Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in UcmTcpciSpec.h.

## Requirements

 Expand table

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT\_BUFFER\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article02/22/2024

Stores the value of the TRANSMIT\_BUFFER Register. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER_IN_PARAMS {
    UCMTCPPIPORTCONTROLLER          PortControllerObject;
    UCMTCPPI_PORT_CONTROLLER_TRANSMIT_BUFFER TransmitBuffer;
} UCMTCPPI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER_IN_PARAMS,
*PUCMTCPPI_PORT_CONTROLLER_SET_TRANSMIT_BUFFER_IN_PARAMS;
```

## Members

PortControllerObject

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

TransmitBuffer

A pointer to the **UCMTCPCI\_PORT\_CONTROLLER\_TRANSMIT\_BUFFER** structure that contains the value to set in the TRANSMIT\_BUFFER Register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in [UcmTcpciSpec.h](#).

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucmtcpciportcontrollerrequests.h

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT\\_BUFFER](#)

# UCMTCPCI\_PORT\_CONTROLLER\_SET\_TRANSMIT\_IN\_PARAMS structure (ucmtcpciportcontrollerrequests.h)

Article 02/22/2024

Stores the values of TRANSMIT Register. This structure is used in the [IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT](#) request.

## Syntax

C++

```
typedef struct _UCMTCPCI_PORT_CONTROLLER_SET_TRANSMIT_IN_PARAMS {
    UCMTCPCHIPORTCONTROLLER          PortControllerObject;
    UCMTCPICI_PORT_CONTROLLER_TRANSMIT Transmit;
} UCMTCPICI_PORT_CONTROLLER_SET_TRANSMIT_IN_PARAMS,
*PUCMTCPICI_PORT_CONTROLLER_SET_TRANSMIT_IN_PARAMS;
```

## Members

`PortControllerObject`

Handle to the port controller object that the client driver received in the previous call to [UcmTcpciPortControllerCreate](#).

`Transmit`

A pointer to the `UCMTCPCI_PORT_CONTROLLER_TRANSMIT` structure that contains the value to set in the TRANSMIT register defined as per the Universal Serial Bus Type-C Port Controller Interface Specification. This structure is declared in `UcmTcpciSpec.h`.

## Requirements

[Expand table](#)

Requirement	Value
Header	<code>ucmtcpciportcontrollerrequests.h</code>

## See also

[IOCTL\\_UCMTCPCI\\_PORT\\_CONTROLLER\\_SET\\_TRANSMIT](#)

# ucmtypes.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucmtypes.h contains the following programming interfaces:

## Functions

<a href="#">UCM_PD_POWER_DATA_OBJECT_GET_TYPE</a>
Retrieves the type of Power Data Object from the UCM_PD_POWER_DATA_OBJECT structure.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_BATTERY</a>
Initializes a UCM_PD_POWER_DATA_OBJECT structure as a Battery Supply type Power Data Object.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_FIXED</a>
Initializes a to the UCM_PD_POWER_DATA_OBJECT for a Fixed Supply type Power Data Object.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT ULONG</a>
Initializes a UCM_PD_POWER_DATA_OBJECT structure by interpreting Power Data Object values and sets each field correctly.
<a href="#">UCM_PD_POWER_DATA_OBJECT_INIT_VARIABLE_NON_BATTERY</a>
Initializes a UCM_PD_POWER_DATA_OBJECT structure as a Variable Supply Non Battery type Power Data Object.
<a href="#">UCM_PD_REQUEST_DATA_OBJECT_INIT ULONG</a>
Initializes a UCM_PD_REQUEST_DATA_OBJECT structure by interpreting Request Data Object values and sets each field correctly.

## Structures

## [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

Describes a Power Data Object. For information about these members, see the Power Delivery specification.

## [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#)

Describes a Request Data Object (RDO). For information about these members, see the Power Delivery specification.

# Enumerations

## [UCM\\_CHARGING\\_STATE](#)

Defines the charging state of a Type-C connector.

## [UCM\\_PD\\_CONN\\_STATE](#)

Defines power delivery (PD) negotiation states of a Type-C port.

## [UCM\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

Defines Power Data Object types.

## [UCM\\_POWER\\_ROLE](#)

Defines power roles of USB Type-C connected devices.

## [UCM\\_TYPEC\\_CURRENT](#)

Defines different Type-C current levels, as defined in the Type-C specification.

## [UCM\\_TYPEC\\_OPERATING\\_MODE](#)

Defines operating modes of a USB Type-C connector.

## [UCM\\_TYPEC\\_PARTNER](#)

Defines the state of the Type-C connector.

# UCM\_CHARGING\_STATE enumeration (ucmtypes.h)

Article02/22/2024

Defines the charging state of a Type-C connector.

## Syntax

C++

```
typedef enum _UCM_CHARGING_STATE {
    UcmChargingStateInvalid,
    UcmChargingStateNotCharging,
    UcmChargingStateNominalCharging,
    UcmChargingStateSlowCharging,
    UcmChargingStateTrickleCharging
} UCM_CHARGING_STATE, *PUCM_CHARGING_STATE;
```

## Constants

[+] Expand table

`UcmChargingStateInvalid`

Indicates the charging state is invalid.

`UcmChargingStateNotCharging`

Indicates the port is not drawing a charge.

`UcmChargingStateNominalCharging`

Indicates the port is drawing a nominal charge.

`UcmChargingStateSlowCharging`

Indicates the port is drawing a slow charge.

`UcmChargingStateTrickleCharging`

Indicates the port is drawing a trickle charge.

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_CONNECTOR\\_PD\\_CONN\\_STATE\\_CHANGED\\_PARAMS](#)

[UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#)

[UcmConnectorPdConnectionStateChanged](#)

[UcmConnectorTypeCAttach](#)

# UCM\_PD\_CONN\_STATE enumeration (ucmtypes.h)

Article02/22/2024

Defines power delivery (PD) negotiation states of a Type-C port.

## Syntax

C++

```
typedef enum _UCM_PD_CONN_STATE {  
    UcmPdConnStateInvalid,  
    UcmPdConnStateNotSupported,  
    UcmPdConnStateNegotiationFailed,  
    UcmPdConnStateNegotiationSucceeded  
} UCM_PD_CONN_STATE;
```

## Constants

[ ] Expand table

<b>UcmPdConnStateInvalid</b> Indicates the PD negotiation state is invalid.
<b>UcmPdConnStateNotSupported</b> Indicates a PD connection is not supported.
<b>UcmPdConnStateNegotiationFailed</b> Indicates the PD negotiation failed.
<b>UcmPdConnStateNegotiationSucceeded</b> Indicates the PD negotiation succeeded.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UcmConnectorPdConnectionStateChanged](#)

# UCM\_PD\_POWER\_DATA\_OBJECT union (ucmtypes.h)

Article04/01/2021

Describes a Power Data Object. For information about these members, see the [Power Delivery specification](#).

## Syntax

C++

```
typedef union _UCM_PD_POWER_DATA_OBJECT {
    ULONG Ul;
    struct {
        unsigned Reserved : 30;
        unsigned Type : 2;
    } Common;
    struct {
        unsigned MaximumCurrentIn10mA : 10;
        unsigned VoltageIn50mV : 10;
        unsigned PeakCurrent : 2;
        unsigned Reserved : 3;
        unsigned DataRoleSwap : 1;
        unsigned UsbCommunicationCapable : 1;
        unsigned ExternallyPowered : 1;
        unsigned UsbSuspendSupported : 1;
        unsigned DualRolePower : 1;
        unsigned FixedSupply : 2;
    } FixedSupplyPdo;
    struct {
        unsigned MaximumCurrentIn10mA : 10;
        unsigned MinimumVoltageIn50mV : 10;
        unsigned MaximumVoltageIn50mV : 10;
        unsigned VariableSupportNonBattery : 2;
    } VariableSupplyNonBatteryPdo;
    struct {
        unsigned MaximumAllowablePowerIn250mW : 10;
        unsigned MinimumVoltageIn50mV : 10;
        unsigned MaximumVoltageIn50mV : 10;
        unsigned Battery : 2;
    } BatterySupplyPdo;
} UCM_PD_POWER_DATA_OBJECT, *PUCM_PD_POWER_DATA_OBJECT;
```

## Members

U1

Size of the structure.

Common

Common.Reserved

Reserved.

Common.Type

Type of Power Data Object.

FixedSupplyPdo

Describing a Fixed Supply type Power Data Object.

FixedSupplyPdo.MaximumCurrentIn10mA

Maximum current in multiples of 10 mA.

FixedSupplyPdo.VoltageIn50mV

Voltage in multiples of 50 mV.

FixedSupplyPdo.PeakCurrent

Peak current.

FixedSupplyPdo.Reserved

Reserved for future use.

FixedSupplyPdo.DataRoleSwap

If set, indicates the Power Data Object can perform a data role swap.

FixedSupplyPdo.UsbCommunicationCapable

If set, indicates the Power Data Object is USB communication capable.

FixedSupplyPdo.ExternallyPowered

If set, indicates the Power Data Object is externally powered.

FixedSupplyPdo.UsbSuspendSupported

Indicates support for USB suspend.

`FixedSupplyPdo.DualRolePower`

Dual role power

`FixedSupplyPdo.FixedSupply`

fixed supply

`VariableSupplyNonBatteryPdo`

Contains bitfields describing a variable-supply non-battery PD object.

`VariableSupplyNonBatteryPdo.MaximumCurrentIn10mA`

Describes the maximum current in multiples of 10 mA.

`VariableSupplyNonBatteryPdo.MinimumVoltageIn50mV`

Describes the minimum voltage in multiples of 50 mV.

`VariableSupplyNonBatteryPdo.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 50 mV.

`VariableSupplyNonBatteryPdo.VariableSupportNonBattery`

Variable Support Non Battery type.

`BatterySupplyPdo`

Contains bitfields describing a battery supply PD object.

`BatterySupplyPdo.MaximumAllowablePowerIn250mW`

Describes the maximum allowable power in multiples of 250 mW.

`BatterySupplyPdo.MinimumVoltageIn50mV`

Describes the minimum voltage in multiples of 50 mV.

`BatterySupplyPdo.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 50 mV.

`BatterySupplyPdo.Battery`

Battery type.

# Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

# UCM\_PD\_POWER\_DATA\_OBJECT\_GET\_TYPE function (ucmtypes.h)

Article 02/22/2024

Retrieves the type of Power Data Object from the [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure.

## Syntax

C++

```
UCM_PD_POWER_DATA_OBJECT_TYPE UCM_PD_POWER_DATA_OBJECT_GET_TYPE(
    [in] PUCM_PD_POWER_DATA_OBJECT Pdo
);
```

## Parameters

[in] Pdo

A pointer to a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure that contains the type of Power Data Object.

## Return value

Returns the `Common.Type` member of the [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure.

## Remarks

For information about the Power Data Object including the types of object, see Power Delivery specification. The `Type` member of [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) indicates the type of Power Data Object.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

# UCM\_PD\_POWER\_DATA\_OBJECT\_INIT\_BATTERY function (ucmtypes.h)

Article02/22/2024

Initializes a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure as a Battery Supply type Power Data Object.

## Syntax

C++

```
void UCM_PD_POWER_DATA_OBJECT_INIT_BATTERY(
    [out] PUCM_PD_POWER_DATA_OBJECT Pdo
);
```

## Parameters

[out] Pdo

A pointer to a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure in which the **BatterySupplyPdo.Battery** member is set to **UcmPdPdoTypeBatterySupply**.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Requirement	Value
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

# UCM\_PD\_POWER\_DATA\_OBJECT\_INIT\_FIXED function (ucmtypes.h)

Article 10/21/2021

Initializes a to the [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) for a Fixed Supply type Power Data Object.

## Syntax

C++

```
void UCM_PD_POWER_DATA_OBJECT_INIT_FIXED(
    [out] PUCM_PD_POWER_DATA_OBJECT Pdo
);
```

## Parameters

[out] Pdo

A pointer to a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure in which the `FixedSupplyPdo.FixedSupply` member is set to `UcmPdPdoTypeFixedSupply`.

## Return value

None

## Remarks

For different types of Power Data Objects, see the power delivery specification.

This function initializes the structure and sets Power Data Object as a Fixed Supply type. The client driver must set the remaining members with values relevant to the specific object type.

## Requirements

<b>Minimum supported client</b>	Windows 10
<b>Minimum supported server</b>	Windows Server 2016
<b>Target Platform</b>	Windows
<b>Minimum KMDF version</b>	1.15
<b>Minimum UMDF version</b>	2.15
<b>Header</b>	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

# UCM\_PD\_POWER\_DATA\_OBJECT\_INIT ULONG function (ucmtypes.h)

Article 02/22/2024

Initializes a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure by interpreting Power Data Object values and sets each field correctly.

## Syntax

C++

```
void UCM_PD_POWER_DATA_OBJECT_INIT ULONG(
    [out] PUCM_PD_POWER_DATA_OBJECT Pdo,
    [in]   ULONG                 UlongInLittleEndian
);
```

## Parameters

[out] Pdo

A pointer to a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure.

[in] UlongInLittleEndian

The ULONG value to set in the UI member of [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#).

## Return value

None

## Remarks

A Power Data Object, as defined by the Power Delivery specification, is a 32-bit value. The hardware is expected to retrieve the Power Data Objects as 32-bit values. This utility function initializes a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure by interpreting those values and setting each field correctly.

The 4 byte value is expected to be in little-endian format. The structure is 4 bytes and the client driver can memcopy the Power Data Objects from the hardware into an array

of [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structures.

# Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

# UCM\_PD\_POWER\_DATA\_OBJECT\_INIT\_VARIABLE\_NON\_BATTERY function (ucmtypes.h)

Article02/22/2024

Initializes a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure as a Variable Supply Non Battery type Power Data Object.

## Syntax

C++

```
void UCM_PD_POWER_DATA_OBJECT_INIT_VARIABLE_NON_BATTERY(
    [out] PUCM_PD_POWER_DATA_OBJECT Pdo
);
```

## Parameters

[out] Pdo

A pointer to a [UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#) structure in which the `VariableSupplyNonBatteryPdo`.`VariableSupportNonBattery` member is set to `UcmPdPdoTypeVariableSupplyNonBattery`.

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT](#)

# UCM\_PD\_POWER\_DATA\_OBJECT\_TYPE enumeration (ucmtypes.h)

Article02/22/2024

Defines Power Data Object types.

## Syntax

C++

```
typedef enum _UCM_PD_POWER_DATA_OBJECT_TYPE {
    UcmPdPdoTypeFixedSupply,
    UcmPdPdoTypeBatterySupply,
    UcmPdPdoTypeVariableSupplyNonBattery
} UCM_PD_POWER_DATA_OBJECT_TYPE;
```

## Constants

[ ] Expand table

<code>UcmPdPdoTypeFixedSupply</code>	Indicates the PD data object type is a fixed supply.
<code>UcmPdPdoTypeBatterySupply</code>	Indicates the PD data object type is a battery supply.
<code>UcmPdPdoTypeVariableSupplyNonBattery</code>	Indicates the PD data object type is a non-battery variable supply.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_POWER\\_DATA\\_OBJECT\\_GET\\_TYPE](#)

# UCM\_PD\_REQUEST\_DATA\_OBJECT union (ucmtypes.h)

Article04/01/2021

Describes a Request Data Object (RDO). For information about these members, see the [Power Delivery specification](#).

## Syntax

C++

```
typedef union _UCM_PD_REQUEST_DATA_OBJECT {
    ULONG U1;
    struct {
        unsigned Reserved1 : 28;
        unsigned ObjectPosition : 3;
        unsigned Reserved2 : 1;
    } Common;
    struct {
        unsigned MaximumOperatingCurrentIn10mA : 10;
        unsigned OperatingCurrentIn10mA : 10;
        unsigned Reserved1 : 4;
        unsigned NoUsbSuspend : 1;
        unsigned UsbCommunicationCapable : 1;
        unsigned CapabilityMismatch : 1;
        unsigned GiveBackFlag : 1;
        unsigned ObjectPosition : 3;
        unsigned Reserved2 : 1;
    } FixedAndVariableRdo;
    struct {
        unsigned MaximumOperatingPowerIn250mW : 10;
        unsigned OperatingPowerIn250mW : 10;
        unsigned Reserved1 : 4;
        unsigned NoUsbSuspend : 1;
        unsigned UsbCommunicationCapable : 1;
        unsigned CapabilityMismatch : 1;
        unsigned GiveBackFlag : 1;
        unsigned ObjectPosition : 3;
        unsigned Reserved2 : 1;
    } BatteryRdo;
} UCM_PD_REQUEST_DATA_OBJECT, *PUCM_PD_REQUEST_DATA_OBJECT;
```

## Members

U1

Size of the structure.

Common

Common.Reserved1

Reserved.

Common.ObjectPosition

Object position.

Common.Reserved2

Reserved.

FixedAndVariableRdo

FixedAndVariableRdo.MaximumOperatingCurrentIn10mA

Maximum current in 10 mA units.

FixedAndVariableRdo.OperatingCurrentIn10mA

Operating current in 10mA units.

FixedAndVariableRdo.Reserved1

Reserved.

FixedAndVariableRdo.NoUsbSuspend

Indicates support for USB suspend.

FixedAndVariableRdo.UsbCommunicationCapable

USB communication capable.

FixedAndVariableRdo.CapabilityMismatch

Capability Mismatch

FixedAndVariableRdo.GiveBackFlag

GiveBack Flag.

FixedAndVariableRdo.ObjectPosition

Object Position.

`BatteryRdo.Reserved2`

Reserved for future use.

`BatteryRdo`

`BatteryRdo.MaximumOperatingPowerIn250mW`

Maximum Operating Power in 250mW units.

`BatteryRdo.OperatingPowerIn250mW`

Operating Power in 250mW units.

`BatteryRdo.Reserved1`

Reserved for future use.

`BatteryRdo.NoUsbSuspend`

USB Suspend.

`BatteryRdo.UsbCommunicationCapable`

USB Communications Capable.

`BatteryRdo.CapabilityMismatch`

Capability Mismatch.

`BatteryRdo.GiveBackFlag`

GiveBack Flag.

`BatteryRdo.ObjectPosition`

Object Position.

`BatteryRdo.Reserved2`

Reserved.

## Requirements

<b>Minimum supported client</b>	Windows 10
<b>Minimum supported server</b>	Windows Server 2016
<b>Minimum KMDF version</b>	1.15
<b>Minimum UMDF version</b>	2.15
<b>Header</b>	ucmtypes.h (include Ucmcx.h)

# UCM\_PD\_REQUEST\_DATA\_OBJECT\_INIT\_ULONG function (ucmtypes.h)

Article04/23/2024

Initializes a [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#) structure by interpreting Request Data Object values and sets each field correctly.

## Syntax

C++

```
void UCM_PD_REQUEST_DATA_OBJECT_INIT ULONG(
    PUCM_PD_REQUEST_DATA_OBJECT Rdo,
    [in] ULONG                 UlongInLittleEndian
);
```

## Parameters

Rdo

A pointer to a [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#) structure.

[in] UlongInLittleEndian

The ULONG value to set in the UI member of [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#).

## Return value

None

## Remarks

For information about Request Data Objects, see the Power Delivery specification. There are different types of Request Data Objects and the type depends on the Power Data Object that is specified in the **ObjectPosition** member of [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#). The source buffer is little-endian format. The client driver can call the [memcpy](#) function to get the Request Data Objects from the hardware into an array of [UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#) structures.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_PD\\_REQUEST\\_DATA\\_OBJECT](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

# UCM\_POWER\_ROLE enumeration (ucmtypes.h)

Article02/22/2024

Defines power roles of USB Type-C connected devices.

## Syntax

C++

```
typedef enum _UCM_POWER_ROLE {
    UcmPowerRoleInvalid,
    UcmPowerRoleSink,
    UcmPowerRoleSource
} UCM_POWER_ROLE;
```

## Constants

[+] Expand table

<code>UcmPowerRoleInvalid</code>	Indicates the power role state is invalid.
<code>UcmPowerRoleSink</code>	Indicates the power role is set to sink power.
<code>UcmPowerRoleSource</code>	Indicates the power role is set to source power.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[EVT\\_UCM\\_CONNECTOR\\_SET\\_POWER\\_ROLE](#)

[UCM\\_CONNECTOR\\_PD\\_CONFIG](#)

[UcmConnectorPowerDirectionChanged](#)

# UCM\_TYPEC\_CURRENT enumeration (ucmtypes.h)

Article 02/22/2024

Defines different Type-C current levels, as defined in the Type-C specification.

## Syntax

C++

```
typedef enum _UCM_TYPEC_CURRENT {
    UcmTypeCCurrentInvalid,
    UcmTypeCCurrentDefaultUsb,
    UcmTypeCCurrent1500mA,
    UcmTypeCCurrent3000mA
} UCM_TYPEC_CURRENT;
```

## Constants

[ ] Expand table

	<b>UcmTypeCCurrentInvalid</b> Indicates the power sourcing current state is invalid.
	<b>UcmTypeCCurrentDefaultUsb</b> Indicates the power sourcing current is the default USB current.
	<b>UcmTypeCCurrent1500mA</b> Indicates the power sourcing current is 1500 mA.
	<b>UcmTypeCCurrent3000mA</b> Indicates the power sourcing current is 3000 mA.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UcmConnectorTypeCAttach](#)

[UcmConnectorTypeCCurrentAdChanged](#)

# UCM\_TYPEC\_OPERATING\_MODE enumeration (ucmtypes.h)

Article02/22/2024

Defines operating modes of a USB Type-C connector.

## Syntax

C++

```
typedef enum _UCM_TYPEC_OPERATING_MODE {  
    UcmTypeCOperatingModeInvalid,  
    UcmTypeCOperatingModeDfp,  
    UcmTypeCOperatingModeUfp,  
    UcmTypeCOperatingModeDrp  
} UCM_TYPEC_OPERATING_MODE;
```

## Constants

[ ] Expand table

	<b>UcmTypeCOperatingModeInvalid</b> Indicates the operating mode is invalid.
	<b>UcmTypeCOperatingModeDfp</b> Indicates the operating mode is set to downstream-facing port.
	<b>UcmTypeCOperatingModeUfp</b> Indicates the operating mode is set to upstream-facing port.
	<b>UcmTypeCOperatingModeDrp</b> Indicates the operating mode is set to dual-role port.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

[UCM\\_CONNECTOR\\_TYPEC\\_CONFIG\\_INIT](#)

# UCM\_TYPEC\_PARTNER enumeration (ucmtypes.h)

Article 02/22/2024

Defines the state of the Type-C connector.

## Syntax

C++

```
typedef enum _UCM_TYPEC_PARTNER {  
    UcmTypeCPartnerInvalid,  
    UcmTypeCPartnerUfp,  
    UcmTypeCPartnerDfp,  
    UcmTypeCPartnerPoweredCableNoUfp,  
    UcmTypeCPartnerPoweredCableWithUfp,  
    UcmTypeCPartnerAudioAccessory,  
    UcmTypeCPartnerDebugAccessory  
} UCM_TYPEC_PARTNER;
```

## Constants

[+] Expand table

<b>UcmTypeCPartnerInvalid</b> The partner port state is invalid.
<b>UcmTypeCPartnerUfp</b> The partner is an upstream facing port (UFP).
<b>UcmTypeCPartnerDfp</b> The partner is a downstream facing port (DFP).
<b>UcmTypeCPartnerPoweredCableNoUfp</b> The partner is a powered cable that requires VConn, that currently does not have a UFP attached on the other end.
<b>UcmTypeCPartnerPoweredCableWithUfp</b> The partner is a powered and upstream facing.
<b>UcmTypeCPartnerAudioAccessory</b> The partner is used as an audio accessory.

#### **UcmTypeCPartnerDebugAccessory**

The partner is a debug accessory.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Minimum UMDF version	2.15
Header	ucmtypes.h (include Ucmcx.h)

## See also

- [UCM\\_CONNECTOR\\_TYPEC\\_ATTACH\\_PARAMS](#)
- [UcmConnectorTypeCAttach](#)

# ucmucsicx.h header

Article01/19/2024

This header is the main include header for client drivers of the UcmUcsiCx class extension. The extension provides a transport-agnostic implementation of the [UCSI specification](#).

Ucmucsicx includes these headers:

- [UcmUcsiGlobals.h](#)
- [UcmUcsiFuncEnum.h](#)
- [UcmUcsiDevice.h](#)
- [UcmUcsiSpec.h](#)
- [UcmUcsiPpm.h](#)
- [UcmUcsiPpmRequests.h](#)

## ⓘ Important

Do not include the preceding headers directly. Instead, only include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)



### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

- ⓘ [Open a documentation issue](#)
- ⓘ [Provide product feedback](#)

# ucmucsdevice.h header

Article04/16/2024

This header provides declaration for functions, callback functions, and structures for a UCM-USCI device.

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsdevice.h contains the following programming interfaces:

## Functions

[+] [Expand table](#)

UCMUCSI_DEVICE_CONFIG_INIT
Initializes a UCMUCSI_DEVICE_CONFIG structure.
<a href="#">UcmUcsiDeviceInitialize</a>
Initializes the UCSI extension (UcmUcsiCx).
<a href="#">UcmUcsiDeviceInitInitialize</a>
Initializes the WDFDEVICE_INIT provided by the framework.

## Structures

[+] [Expand table](#)

UCMUCSI_DEVICE_CONFIG
Configuration structure for UcmUcsiDeviceInitialize.

---

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# UCMUCSI\_DEVICE\_CONFIG structure (ucmucsdevice.h)

Article02/22/2024

Configuration structure for [UcmUcsiDeviceInitialize](#) that initializes a framework device object.

## Syntax

C++

```
typedef struct _UCMUCSI_DEVICE_CONFIG {
    ULONG Size;
} UCMUCSI_DEVICE_CONFIG, *PUCMUCSI_DEVICE_CONFIG;
```

## Members

Size

Size of this structure.

## Requirements

  [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsdevice.h (include UcmUcsiCx.h)

## See also

[UcmUcsiDeviceInitialize](#)

[EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#)

[UCMUCSI\\_DEVICE\\_CONFIG\\_INIT](#)

## **WdfDeviceCreate**

# UCMUCSI\_DEVICE\_CONFIG\_INIT function (ucmucsdevice.h)

Article02/22/2024

Initializes a [UCMUCSI\\_DEVICE\\_CONFIG](#) structure.

## Syntax

C++

```
void UCMUCSI_DEVICE_CONFIG_INIT(
    PUCMUCSI_DEVICE_CONFIG Config
);
```

## Parameters

Config

A pointer to the [UCMUCSI\\_DEVICE\\_CONFIG](#) structure to initialize.

## Return value

None

## Remarks

The client driver must call this initialization function before calling [UcmUcsiDeviceInitialize](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsdevice.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib

## See also

[EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#)

[UCMUCSI\\_DEVICE\\_CONFIG](#)

[WdfDeviceCreate](#)

# UcmUcsiDeviceInitialize function (ucmucsiedevice.h)

Article02/22/2024

Initializes the UCSI extension (UcmUcsiCx).

## Syntax

C++

```
NTSTATUS UcmUcsiDeviceInitialize(
    WDFDEVICE             WdfDevice,
    PUCMUCSI_DEVICE_CONFIG Config
);
```

## Parameters

`WdfDevice`

A handle to a framework device object that the client driver received in a previous call to [WdfDeviceCreate](#).

`Config`

A pointer to a caller-supplied [UCMUCSI\\_DEVICE\\_CONFIG](#) structure that is initialized by calling [UCMUCSI\\_DEVICE\\_CONFIG\\_INIT](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS code.

## Remarks

The client driver must call `UcmUcsiDeviceInitialize` in the driver's [EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#) implementation after calling [WdfDeviceCreate](#) successfully.

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsdevice.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD](#)

[UCMUCSI\\_DEVICE\\_CONFIG](#)

[UCMUCSI\\_DEVICE\\_CONFIG\\_INIT](#)

[WdfDeviceCreate](#)

# UcmUcsiDeviceInitInitialize function (ucmucsdevice.h)

Article 02/22/2024

Initializes the [WDFDEVICE\\_INIT](#) provided by the framework.

## Syntax

C++

```
NTSTATUS UcmUcsiDeviceInitInitialize(
    PWDFDEVICE_INIT DeviceInit
);
```

## Parameters

`DeviceInit`

A pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS code.

## Remarks

The client driver must call this function after calling [WdfDeviceInitSetPnpPowerEventCallbacks](#). This function initializes the UCSI extension (UcmUcsiCx) with the framework [WDFDEVICE\\_INIT](#) structure that contains pointers to PnP and power callback functions implemented by the client driver.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsdevice.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib

# ucmucsifuncenum.h header

Article01/23/2023

This header declares an enumeration of all export functions called by a client driver of a UcmUcsiCx class extension.

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsifuncenum.h contains the following programming interfaces:

## Enumerations

### UCMUCSIFUNCENUM

Defines values for all export functions called by a client driver of a UcmUcsiCx class extension.

# UCMUCSIFUNCENUM enumeration (ucmucsifuncenum.h)

Article 06/03/2021

Defines values for all export functions called by a client driver of a UcmUcsiCx class extension.

## Syntax

C++

```
typedef enum _UCMUCSIFUNCENUM {
    UcmUcsiDeviceInitInitializeTableIndex,
    UcmUcsiDeviceInitializeTableIndex,
    UcmUcsiConnectorCollectionCreateTableIndex,
    UcmUcsiConnectorCollectionAddConnectorTableIndex,
    UcmUcsiPpmCreateTableIndex,
    UcmUcsiPpmSetUcsiCommandRequestQueueTableIndex,
    UcmUcsiPpmStartTableIndex,
    UcmUcsiPpmStopTableIndex,
    UcmUcsiPpmNotificationTableIndex,
    UcmUcsiFunctionTableNumEntries
} UCMUCSIFUNCENUM;
```

## Constants

`UcmUcsiDeviceInitInitializeTableIndex`

UcmUcsiDeviceInitInitialize - initializes the `WDFDEVICE_INIT` provided by the framework.

`UcmUcsiDeviceInitializeTableIndex`

UcmUcsiDeviceInitialize - initializes the UcmUcsiCx class extension.

`UcmUcsiConnectorCollectionCreateTableIndex`

UcmUcsiConnectorCollectionCreate - creates a connector collection object with UcmUcsiCx.

`UcmUcsiConnectorCollectionAddConnectorTableIndex`

UcmUcsiConnectorCollectionAddConnector - adds a connector to the connector collection object.

`UcmUcsiPpmCreateTableIndex`

UcmUcsiPpmCreate - creates a Platform Policy Manager (PPM) object.

**`UcmUcsiPpmSetUcsiCommandRequestQueueTableIndex`**

`UcmUcsiPpmSetUcsiCommandRequestQueue` - provides a framework queue object that is used to dispatch UCSI commands to the client driver.

**`UcmUcsiPpmStartTableIndex`**

`UcmUcsiPpmStart` - instructs the class extension to start sending requests to the client driver.

**`UcmUcsiPpmStopTableIndex`**

`UcmUcsiPpmStop` - instructs the class extension to stop sending requests to the client driver.

**`UcmUcsiPpmNotificationTableIndex`**

`UcmUcsiPpmNotification` - informs the `UcmUcsiCx` class extension about a UCSI notification.

**`UcmucsifunctionTableNumEntries`**

End of this enumeration.

## Requirements

**Minimum KMDF version** 1.27

**Minimum UMDF version** N/A

**Header** `ucmucsifuncenum.h` (include `UcmUcsiCx.h`)

## See also

[Ucmucsicx.h](#)

# ucmucsiglobals.h header

Article01/23/2023

This header provides UCM-UCSI global definitions.

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsiglobals.h contains the following programming interfaces:

## Structures

[\*\*UCMUCSI\\_DRIVER\\_GLOBALS\*\*](#)

Reserved for UCMUCSI\_DRIVER\_GLOBALS.

# UCMUCSI\_DRIVER\_GLOBALS structure (ucmucsiglobals.h)

Article02/22/2024

Reserved.

## Syntax

C++

```
typedef struct _UCMUCSI_DRIVER_GLOBALS {
    ULONG Reserved;
} UCMUCSI_DRIVER_GLOBALS, *PUCMUCSI_DRIVER_GLOBALS;
```

## Members

Reserved

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsiglobals.h

## See also

[Ucmucsicx.h](#)

# ucmucsippm.h header

Article01/23/2023

This header provides declarations for UCM-UCSI Platform Policy Manager (PPM) abstraction within the class extension. This PPM object implements the details of sending UCSI commands from Operating System Policy Manager (OPM) object to the client driver and receiving notifications from the client driver. For sending commands to the client driver, it converts UCSI PPM commands to IOCTLs and forwards them to the client driver which later transports the commands to the actual firmware.

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsippm.h contains the following programming interfaces:

## Functions

<a href="#">UCMUCSI_CONNECTOR_INFO_INIT</a>
Initializes a UCMUCSI_CONNECTOR_INFO structure.
<a href="#">UCMUCSI_PPM_CONFIG_INIT</a>
Initializes a UCMUCSI_PPM_CONFIG structure.
<a href="#">UcmUcsiConnectorCollectionAddConnector</a>
Adds a connector to the connector collection object.
<a href="#">UcmUcsiConnectorCollectionCreate</a>
Creates a connector collection object with UcmUcsiCx.
<a href="#">UcmUcsiPpmCreate</a>
Creates a Platform Policy Manager (PPM) object.
<a href="#">UcmUcsiPpmNotification</a>

Informs the UcmUcsiCx class extension about a UCSI notification.

#### [UcmUcsiPpmSetUcsiCommandRequestQueue](#)

Provides a framework queue object that is used to dispatch UCSI commands to the client driver.

#### [UcmUcsiPpmStart](#)

Instructs the class extension to start sending requests to the client driver.

#### [UcmUcsiPpmStop](#)

Instructs the class extension to stop sending requests to the client driver.

## Structures

#### [UCMUCSI\\_CONNECTOR\\_INFO](#)

Stores information about connectors that cannot be obtained by sending UCSI commands such as "Get Connector Capability".

#### [UCMUCSI\\_PPM\\_CONFIG](#)

Stores configuration information required to create a Platform Policy Manager (PPM).

# UCMUCSI\_CONNECTOR\_INFO structure (ucmucsippm.h)

Article02/22/2024

Stores information about connectors that cannot be obtained by sending UCSI commands such as GetConnectorCapability. Initialize this structure by calling [UCMUCSI\\_CONNECTOR\\_INFO\\_INIT](#). This structure is used in the [UcmUcsiConnectorCollectionAddConnector](#) call.

## Syntax

C++

```
typedef struct _UCMUCSI_CONNECTOR_INFO {
    ULONG      Size;
    ULONGLONG ConnectorId;
} UCMUCSI_CONNECTOR_INFO, *PUCMUCSI_CONNECTOR_INFO;
```

## Members

Size

Size of this structure.

ConnectorId

Connector ID that maps a USB Type-C connector to USB port.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippm.h (include UcmUcsiCx.h)

# UCMUCSI\_CONNECTOR\_INFO\_INIT function (ucmucsippm.h)

Article02/22/2024

Initializes a [UCMUCSI\\_CONNECTOR\\_INFO](#) structure.

## Syntax

C++

```
void UCMUCSI_CONNECTOR_INFO_INIT(
    [out] PUCMUCSI_CONNECTOR_INFO ConnectorInfo
);
```

## Parameters

[out] ConnectorInfo

A pointer to a [UCMUCSI\\_CONNECTOR\\_INFO](#) structure to initialize.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippm.h (include UcmUcsiCx.h)

## See also

[UCMUCSI\\_CONNECTOR\\_INFO](#)

## UcmUcsiConnectorCollectionAddConnector

# UCMUCSI\_PPM\_CONFIG structure (ucmucsippm.h)

Article02/22/2024

Stores configuration information required to create a Platform Policy Manager (PPM). Initialize this structure by calling [UCMUCSI\\_PPM\\_CONFIG\\_INIT](#). This structure is used in the [UcmUcsiPpmCreate](#) call.

## Syntax

C++

```
typedef struct _UCMUCSI_PPM_CONFIG {
    ULONG             Size;
    BOOLEAN           UsbDeviceControllerEnabled;
    UCMUCSI_CONNECTOR_COLLECTION ConnectorCollectionHandle;
} UCMUCSI_PPM_CONFIG, *PUCMUCSI_PPM_CONFIG;
```

## Members

Size

UsbDeviceControllerEnabled

A boolean value that indicates whether or not to enable the device controller.

ConnectorCollectionHandle

The **ConnectorCollectionHandle** member must be set to the handle retrieved in a previous call to [UcmUcsiConnectorCollectionCreate](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsippm.h (include UcmUcsiCx.h)

## See also

[UcmUcsiPpmCreate](#) [UcmUcsiConnectorCollectionCreate](#)

# UCMUCSI\_PPM\_CONFIG\_INIT function (ucmucsippm.h)

Article02/22/2024

Initializes a [UCMUCSI\\_PPM\\_CONFIG](#) structure.

## Syntax

C++

```
void UCMUCSI_PPM_CONFIG_INIT(
    PUCMUCSI_PPM_CONFIG          Config,
    UCMUCSI_CONNECTOR_COLLECTION CollectionObject
);
```

## Parameters

Config

A pointer to a [UCMUCSI\\_PPM\\_CONFIG](#) structure to initialize.

CollectionObject

The handle to the connector collection object that the client driver retrieved in a previous call to [UcmUcsiConnectorCollectionCreate](#).

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsippm.h (include UcmUcsiCx.h)

## See also

[UcmUcsiPpmCreate](#) [UcmUcsiConnectorCollectionCreate](#)

# UcmUcsiConnectorCollectionAddConnector function (Ucmucsippm.h)

Article02/22/2024

Adds a connector to the connector collection object.

## Syntax

C++

```
NTSTATUS UcmUcsiConnectorCollectionAddConnector(
    [in] UCMUCSI_CONNECTOR_COLLECTION ConnectorCollectionObject,
    [in] PUCMUCSI_CONNECTOR_INFO      ConnectorInfo
);
```

## Parameters

[in] ConnectorCollectionObject

The handle to the connector collection object that the client driver retrieved in a previous call to [UcmUcsiConnectorCollectionCreate](#).

[in] ConnectorInfo

A pointer to a [UCMUCSI\\_CONNECTOR\\_INFO](#) structure that contains information about the connector to add.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

The client driver must not call [UcmUcsiConnectorCollectionAddConnector](#) after [UcmUcsiPpmCreate](#) because it would have no effect on the already existing PPM object.

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	Ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[UCMUCSI\\_CONNECTOR\\_INFO](#)

[UcmUcsiConnectorCollectionCreate](#)

[UcmUcsiPpmCreate](#)

# UcmUcsiConnectorCollectionCreate function (ucmucsippm.h)

Article02/22/2024

Creates a connector collection object with UcmUcsiCx.

## Syntax

C++

```
NTSTATUS UcmUcsiConnectorCollectionCreate(
    [in] WDFDEVICE                  WdfDevice,
    [in] PWDF_OBJECT_ATTRIBUTES     Attributes,
    [out] UCMUCSI_CONNECTOR_COLLECTION *ConnectorCollection
);
```

## Parameters

[in] WdfDevice

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

[in] Attributes

A pointer to a caller-supplied [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that contains attributes for the new connector collection object. This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

[out] ConnectorCollection

A pointer to a location that receives a handle to the new connector collection object.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate [NTSTATUS](#) value.

## Remarks

The collection object is required for creating a Platform Policy Manager (PPM) object. The client driver creates the object by calling [UcmUcsiPpmCreate](#). The driver must not call [UcmUcsiConnectorCollectionCreate] after UcmUcsiPpmCreate because it would have no effect on the already existing PPM object.

The connector collection object is parented to the WDFOBJECT even when UcmUcsiConnectorCollectionCreate is called by passing WDF\_NO\_ATTRIBUTES. The lifetime of the object is manager by the framework.

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[WdfDeviceCreate](#)

[WDF\\_OBJECT\\_ATTRIBUTES](#)

[UcmUcsiPpmCreate](#)

[UcmUcsiConnectorCollectionAddConnector](#)

# UcmUcsiPpmCreate function (Ucmucsippm.h)

Article02/22/2024

Creates a Platform Policy Manager (PPM) object.

## Syntax

C++

```
NTSTATUS UcmUcsiPpmCreate(
    [in] WDFDEVICE             WdfDevice,
    [in] PUCMUCSI_PPM_CONFIG   Config,
    [in] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out] UCMUCSIPPM          *PPMObject
);
```

## Parameters

[in] WdfDevice

A handle to a framework device object that the client driver received in the previous call to [WdfDeviceCreate](#).

[in] Config

A pointer to a caller-supplied [UCMUCSI\\_PPM\\_CONFIG](#) structure that is initialized by calling [UCMUCSI\\_PPM\\_CONFIG\\_INIT](#). The **ConnectorCollectionHandle** member must be set to the handle retrieved in a previous call to [UcmUcsiConnectorCollectionCreate](#).

[in] Attributes

A pointer to a caller-supplied [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that contains attributes for the new connector collection object. This parameter is optional and can be WDF\_NO\_OBJECT\_ATTRIBUTES.

[out] PPMObject

A pointer to a location that receives a handle to the new PPM object.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate NTSTATUS value.

## Remarks

The client driver is expected to call **UcmUcsiPpmCreate** from the **EVT\_WDF\_DEVICE\_PREPARE\_HARDWARE** callback function.

The structure passed in *Config* contains Type-C connector information that is necessary for creating connectors with USB Type-C connector class extension (UcmCx) by using **UcmConnectorCreate** and are not obtainable from PPM through UCSI commands such as GetCapability or GetConnectorCapability.

The *Config* structure also contains connector IDs, which are required for one-to-one mapping between USB Type-C connectors and USB ports.

The PPM object is also a WDFOBJECT and creates a one-to-one association with the WDFDEVICE handle provided by the client driver.

The PPM object is parented to the WDFOBJECT even when UcmUcsiPpmCreate is called by passing WDF\_NO\_ATTRIBUTES. The lifetime of the object is manager by the framework.

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	Ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

# UcmUcsiPpmNotification function (ucmucsippm.h)

Article 02/22/2024

Informs the UcmUcsiCx class extension about a UCSI notification.

## Syntax

C++

```
void UcmUcsiPpmNotification(
    [in] UCMUCSIPPM      PpmObject,
    [in] PUCSI_DATA_BLOCK DataBlock
);
```

## Parameters

[in] PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

[in] DataBlock

A pointer to a [UCSI\_DATA\_BLOCK] structure that contains information about the USCI notification.

## Return value

None

## Remarks

The client driver calls UcmUcsiPpmNotification in the event of a UCSI notification. The driver must not call more than one instance of this function at the same time to avoid a race condition.

The client driver should assume that the driver can receive a request before this call returns. Hence, if it keeps a lock around this function and the same lock around

handling a request, it will result into a deadlock.

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	<=DISPATCH_LEVEL

# UcmUcsiPpmSetUcsiCommandRequestQueue function (Ucmucsippm.h)

Article02/22/2024

Provides a framework queue object that is used to dispatch UCSI commands to the client driver.

## Syntax

C++

```
void UcmUcsiPpmSetUcsiCommandRequestQueue(
    [in] UCMUCSIPPM PpmObject,
    [in] WDFQUEUE    PpmRequestQueue
);
```

## Parameters

[in] PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

[in] PpmRequestQueue

A WDFQUEUE handle that the client driver in a previous call to [WdfIoQueueCreate](#)

## Return value

None

## Remarks

The client driver is expected to call UcmUcsiPpmSetUcsiCommandRequestQueue after calling [UcmUcsiPpmCreate](#) and before [UcmUcsiPpmStart](#).

## Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	Ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

# UcmUcsiPpmStart function (Ucmucsippm.h)

Article 10/21/2021

Instructs the UcmUcsiCx class extension to start sending requests to the client driver.

## Syntax

C++

```
NTSTATUS UcmUcsiPpmStart(  
    [in] UCMUCSIPPMM PpmObject  
);
```

## Parameters

[in] PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method can return an appropriate NTSTATUS value.

## Remarks

**UcmUcsiPpmStart** indicates that the client driver is now ready to receive request from the class extension. Upon this call, the class extension starts OS Policy Manager (OPM) and Command Handler state machines.

The client driver must call **UcmUcsiPpmStart** after it had called **UcmUcsiPpmStop** for error recovery.

This DDI starts the operations that the class extension needs to perform to initialize the OPM and Command Handler state machines. The client driver must call **UcmUcsiPpmStart** to notify UcmUcsiCx that the driver is ready to receive the IOCTL requests. We recommend that you make this call either from the

[EVT\\_WDF\\_DEVICE\\_PREPARE\\_HARDWARE](#) callback function, or after the system calls this callback.

Attempting to start the PPM after it has already started leads to an error condition.

After the client calls [UcmUcsiPpmStart](#), the class extension sends a number of commands to the PPM firmware in order to get PPM and connector capabilities and their statuses. Due to a high number interactions with the firmware, we strongly recommend UcmUcsiCx client implementors call this DDI once during startup and not on resume from a low power state, such as D0Entry callback. This is especially true when the client implements S0 idling.

## Requirements

Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	Ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

# UcmUcsiPpmStop function (ucmucsippm.h)

Article02/22/2024

Instructs the UcmUcsiCx class extension to stop sending requests to the client driver.

## Syntax

C++

```
void UcmUcsiPpmStop(
    [in] UCMUCSIPPM PpmObject
);
```

## Parameters

[in] PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

## Return value

None

## Remarks

**UcmUcsiPpmStop** indicates that the client driver is no longer ready to receive requests from the class extension. The class extension guarantees that there will not be any requests made to the client after this call returns. The driver should call this DDI when it encounters a fault and wants the class extension to stop sending PPM requests. After the call completes, the driver should start the PPM again using [UcmUcsiPpmStart](#).

The client driver is expected to call this DDI on driver unload. This call indicates the class extension to start tearing down its internal state machines. It is recommended that the client calls **UcmUcsiPpmStop** from its EVT\_WDF\_DEVICE\_RELEASE\_HARDWARE callback.

Because **UcmUcsiPpmStop** relies on sending UCSI commands to PPM over the power-managed WDFQUEUE provided by the client driver, an attempt to call this function from

[EVT\\_WDF\\_DEVICE\\_D0\\_EXIT](#) callback results in a failure. That is because at this time, the dispatch gates for the queue are closed.

After the [UcmUcsiPpmStop](#) returns, [UcmUcsiPpmStart](#) can be called to start the PPM again.

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippm.h (include UcmUcsiCx.h)
Library	UcmUcsiCxStub.lib
IRQL	PASSIVE_LEVEL

# ucmucsippmrequests.h header

Article 01/23/2023

UCM-UCSI Platform Policy Manager (PPM) abstracts the details of sending UCSI commands from Operating System Policy Manager (OPM) to PPM and receiving notifications from the PPM. It converts PPM commands to WDFREQUEST objects and forwards them to the client driver. UCSI commands are sent to the client driver as I/O control codes, declared in this header.

For information about UCSI commands, see [UCSI spec version 1.1 ↗](#).

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsippmrequests.h contains the following programming interfaces:

## IOCTLs

[IOCTL\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#)

Learn more about: IOCTL\_UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK IOCTL

[IOCTL\\_UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK](#)

Sends a UCSI data block to the client driver.

## Structures

[UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#)

Contains a UCSI data block for input to IOCTL\_UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK.

[UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_OUT\\_PARAMS](#)

Contains a USCI data block for output to IOCTL\_UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK.

#### [UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#)

Contains a USCI data block for input to IOCTL\_UCMUCSI\_PPM\_SEND\_UCSI\_DATA\_BLOCK.

## Enumerations

#### [UCMUCSI\\_PPM\\_IOCTL](#)

Defines I/O control codes handled by the client driver.

# **IOCTL\_UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK IOCTL (Ucmucsippmrequests.h)**

Article02/22/2024

Gets a UCSI data block from the PPM firmware by using the supported transport.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer a [UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#) structure that contains the PPM object that managers the PPM hardware.

## **Input buffer length**

Size of the [UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#) structure.

## **Output buffer**

A pointer a [UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_OUT\\_PARAMS](#) structure that contains the PPM object that managers the PPM hardware.

## **Output buffer length**

Size of the [UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK\\_OUT\\_PARAMS](#) structure.

## **Status block**

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful. Otherwise, set to the appropriate error condition as a NTSTATUS code. For more information, see [NTSTATUS Values](#).

## **Remarks**

Certain UCSI commands do not generate notifications from the PPM firmware, such as the PPM\_RESET command. When this command is received, the firmware disables all

notification. The UcmUcsiCx class extension sends such commands to the client driver through this IOCTL request. The client driver is expected to poll on reset complete indicator and return the current result from the firmware to UcmUcsiCx.

## Requirements

 Expand table

Requirement	Value
Header	Ucmucsippmrequests.h (include UcmUcsiCx.h)

# **IOCTL\_UCMUCSI\_PPM\_SEND\_UCSI\_DATA\_BLOCK IOCTL (Ucmucsippmrequests.h)**

Article02/22/2024

Sends a UCSI data block to the PPM firmware by using the supported transport.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#) structure that contains the UCSI data block.

## **Input buffer length**

Size of the [UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#) structure.

## **Status block**

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful. Otherwise, set to the appropriate error condition as a NTSTATUS code. For more information, see [NTSTATUS Values](#).

## **Remarks**

Whenever the UcmUcsiCx class extension asynchronously needs to send a UCSI block to the PPM firmware, the class extension sends this IOCTL request to the client driver asynchronously.

## **Requirements**

[\[\] Expand table](#)

Requirement	Value
Header	Ucmucsippmrequests.h (include UcmUcsiCx.h)

## See also

[UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK\\_IN\\_PARAMS](#)

# UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK\_IN\_PARAMS structure (ucmucsippmrequests.h)

Article02/22/2024

Contains a UCSI data block. This structure is the input buffer to the [IOCTL\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#) I/O request.

## Syntax

C++

```
typedef struct _UCMUCSI_PPM_GET_UCSI_DATA_BLOCK_IN_PARAMS {
    UCMUCSIPPM PpmObject;
} UCMUCSI_PPM_GET_UCSI_DATA_BLOCK_IN_PARAMS,
*PUCMUCSI_PPM_GET_UCSI_DATA_BLOCK_IN_PARAMS;
```

## Members

### PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippmrequests.h (include UcmUcsiCx.h)

## See also

[IOCTL\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#)

# UCMUCSI\_PPM\_GET\_UCSI\_DATA\_BLOCK\_OUT\_PARAMS structure (ucmucsippmrequests.h)

Article02/22/2024

Contains a UCSI data block. This structure is the output buffer to the [IOCTL\\_UCMUCSI\\_PPM\\_GET\\_UCSI\\_DATA\\_BLOCK](#) I/O request.

## Syntax

C++

```
typedef struct _UCMUCSI_PPM_GET_UCSI_DATA_BLOCK_OUT_PARAMS {
    UCSI_DATA_BLOCK UcmUcsiDataBlock;
} UCMUCSI_PPM_GET_UCSI_DATA_BLOCK_OUT_PARAMS,
*PUCMUCSI_PPM_GET_UCSI_DATA_BLOCK_OUT_PARAMS;
```

## Members

`UcmUcsiDataBlock`

A [UCSI\\_DATA\\_BLOCK](#) structure that receives the data block with notification information.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippmrequests.h (include UcmUcsiCx.h)

## See also

[IOCTL\\_UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK](#)

# UCMUCSI\_PPM\_IOCTL enumeration (ucmucsippmrequests.h)

Article02/22/2024

## Syntax

C++

```
typedef enum _UCMUCSI_PPM_IOCTL {
    _IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK,
    _IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK
} UCMUCSI_PPM_IOCTL;
```

## Constants

[ ] Expand table

<code>_IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK</code> See <a href="#">IOCTL_UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK</a> .
<code>_IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK</code> See <a href="#">IOCTL_UCMUCSI_PPM_GET_UCSI_DATA_BLOCK</a>

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippmrequests.h (include UcmUcsiCx.h)

# UCMUCSI\_PPM\_SEND\_UCSI\_DATA\_BLOCK\_IN\_PARAMS structure (ucmucsippmrequests.h)

Article02/22/2024

Contains a UCSI data block. This structure is the input buffer to the [IOCTL\\_UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK](#) I/O request.

## Syntax

C++

```
typedef struct _UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK_IN_PARAMS {
    UCMUCSIPPM      PpmObject;
    UCSI_DATA_BLOCK UcmUcsiDataBlock;
} UCMUCSI_PPM_SEND_UCSI_DATA_BLOCK_IN_PARAMS,
*PUCMUCSI_PPM_SEND_UCSI_DATA_BLOCK_IN_PARAMS;
```

## Members

PpmObject

A handle to a Platform Policy Manager (PPM) object that the client driver received in the previous call to [UcmUcsiPpmCreate](#).

UcmUcsiDataBlock

A [UCSI\\_DATA\\_BLOCK](#) structure that contains the data block to send.

## Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsippmrequests.h (include UcmUcsiCx.h)

## See also

[IOCTL\\_UCMUCSI\\_PPM\\_SEND\\_UCSI\\_DATA\\_BLOCK](#)

# ucmucsispes.h header

Article05/22/2024

This header declares registers defined in the Intel UCSI Specification 1.1.

For information, see [UCSI spec version 1.2](#).

Do not include this header. Instead, include Ucmucsicx.h.

For more information, see:

- [Write a UcmUcsi client driver](#)
- [Universal Serial Bus \(USB\)](#)

ucmucsispes.h contains the following programming interfaces:

## Functions

[Expand table](#)

### [UCSI\\_CMD\\_SUCCEEDED](#)

On successful completion of a UCSI command the PPM firmware fills the CCI Data Structure provided by the client driver.

## Structures

[Expand table](#)

### [UCSI\\_ACK\\_CC\\_CI\\_COMMAND](#)

Used in the ACK\_CC\_CI command. See Table 4-7.

### [UCSI\\_ALTERNATE\\_MODE](#)

Used in GET\_ALTERNATE\_MODES command. See Table 4-26.

### [UCSI\\_BM\\_POWER\\_SOURCE](#)

	Used in GET_CAPABILITY command. See Bit 15:8 in Table 4-14.
<a href="#">UCSI_CCI</a>	Used in GET_CONNECTOR_CAPABILITY command. See Table 4-16.
<a href="#">UCSI_CONNECTOR_RESET_COMMAND</a>	Used in the CONNECTOR_RESET command. See Table 4-5.
<a href="#">UCSI_CONTROL</a>	Used in the SET_NOTIFICATION_ENABLE command. See Table 4-9.
<a href="#">UCSI_DATA_BLOCK</a>	The data structures for memory locations. See Section 3.
<a href="#">UCSI_GET_ALTERNATE_MODES_COMMAND</a>	Used in the GET_ALTERNATE_MODES command. See Table 4-24.
<a href="#">UCSI_GET_ALTERNATE_MODES_IN</a>	Learn how UCSI_GET_ALTERNATE_MODES_IN is used in the GET_ALTERNATE_MODES command. See Table 4-24.
<a href="#">UCSI_GET_CABLE_PROPERTY_COMMAND</a>	Used in the GET_CABLE_PROPERTY command. See Table 4-37.
<a href="#">UCSI_GET_CABLE_PROPERTY_IN</a>	Used in the GET_CABLE_PROPERTY command. See Table 4-39.
<a href="#">UCSI_GET_CAM_SUPPORTED_COMMAND</a>	Used in the GET_CAM_SUPPORTED command. See Table 4-27.
<a href="#">UCSI_GET_CAM_SUPPORTED_IN</a>	Learn how UCSI_GET_CAM_SUPPORTED_IN is used in the GET_CAM_SUPPORTED command. See Table 4-27.
<a href="#">UCSI_GET_CAPABILITY_IN</a>	Used in the GET_CAPABILITY command. See Table 4-13.
<a href="#">UCSI_GET_CONNECTOR_CAPABILITY_COMMAND</a>	

Used in the GET\_CONNECTOR\_CAPABILITY command. See Table 4-15.

#### [UCSI\\_GET\\_CONNECTOR\\_CAPABILITY\\_IN](#)

Used in the GET\_CONNECTOR\_CAPABILITY command.

#### [UCSI\\_GET\\_CONNECTOR\\_STATUS\\_COMMAND](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-40.

#### [UCSI\\_GET\\_CONNECTOR\\_STATUS\\_IN](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42.

#### [UCSI\\_GET\\_CURRENT\\_CAM\\_COMMAND](#)

Used in the GET\_CURRENT\_CAM command. See Table 4-29.

#### [UCSI\\_GET\\_CURRENT\\_CAM\\_IN](#)

Used in the GET\_CURRENT\_CAM command. See Table 4-31.

#### [UCSI\\_GET\\_ERROR\\_STATUS\\_COMMAND](#)

Used in the GET\_ERROR\_STATUS command. See Table 4-45

#### [UCSI\\_GET\\_ERROR\\_STATUS\\_IN](#)

Used in the GET\_ERROR\_STATUS command. See Table 4-47.

#### [UCSI\\_GET\\_PDOS\\_COMMAND](#)

Used in the GET\_PDOS command. See Table 4-34.

#### [UCSI\\_GET\\_PDOS\\_IN](#)

Used in the GET\_PDOS command. See Table 4-36.

#### [UCSI\\_MESSAGE\\_IN](#)

The MESSAGE IN data structure. See Section 3.4.

#### [UCSI\\_MESSAGE\\_OUT](#)

The MESSAGE OUT data structure. See Section 3.5.

#### [UCSI\\_SET\\_NEW\\_CAM\\_COMMAND](#)

Used in the SET\_NEW\_CAM command. See Table 4-32.

## [UCSI\\_SET\\_NOTIFICATION\\_ENABLE\\_COMMAND](#)

Learn how UCSI\_SET\_NOTIFICATION\_ENABLE\_COMMAND is used in the SET\_NOTIFICATION\_ENABLE command. See Table 4-9.

## [UCSI\\_SET\\_PDM\\_COMMAND](#)

\_UCSI\_SET\_PDM\_COMMAND is obsolete.

## [UCSI\\_SET\\_PDR\\_COMMAND](#)

Used in the SET\_PDR command. See Table 4-22.

## [UCSI\\_SET\\_POWER\\_LEVEL\\_COMMAND](#)

Used in the SET\_POWER\_LEVEL command. See Table 4-48.

## [UCSI\\_SET\\_UOM\\_COMMAND](#)

Used in the SET\_UOM command. See Table 4-18.

## [UCSI\\_SET\\_UOR\\_COMMAND](#)

Used in the SET\_UOR command. See Table 4-20.

## [UCSI\\_VERSION](#)

The VERSION data structure. See Section 3.1.

# Enumerations

[\[+\] Expand table](#)

## [UCSI\\_BATTERY\\_CHARGING\\_STATUS](#)

See Table 4-42, Offset 64.

## [UCSI\\_COMMAND](#)

See Table 4-51, Command Code.

## [UCSI\\_CONNECTOR\\_PARTNER\\_FLAGS](#)

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 21.

<a href="#">UCSI_CONNECTOR_PARTNER_TYPE</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 29.
<a href="#">UCSI_GET_ALTERNATE_MODES_RECIPIENT</a>	Used in the GET_ALTERNATE_MODES command. See Table 4-24, Offset 16.
<a href="#">UCSI_GET_PDOS_SOURCE_CAPABILITIES_TYPE</a>	Used in the GET_PDOS command. See Table 4-34, Offset 35.
<a href="#">UCSI_GET_PDOS_TYPE</a>	Used in the GET_PDOS command. See Table 4-34, Offset 34.
<a href="#">UCSI_POWER_DIRECTION</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 20.
<a href="#">UCSI_POWER_DIRECTION_MODE</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 20.
<a href="#">UCSI_POWER_DIRECTION_ROLE</a>	Used in the SET_PDR command. The SET_PDR command is used to set the power direction dictated by the OS Policy Manager (OPM), for the current connection.
<a href="#">UCSI_POWER_OPERATION_MODE</a>	Used in the GET_CONNECTOR_STATUS command. See Table 4-42, Offset 16.
<a href="#">UCSI_USB_OPERATION_MODE</a>	Used in the SET_UOR command. See Table 4-18, Offset 23.
<a href="#">UCSI_USB_OPERATION_ROLE</a>	Used in the SET_UOR command. The SET_UOR command is used to set the USB operation role dictated by the OS Policy Manager (OPM), for the current connection.

## Feedback

Was this page helpful?



Yes



No



# UCSI\_ACK\_CC\_CI\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Used in the ACK\_CC\_CI command. See Table 4-7 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_ACK_CC_CI_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorChangeAcknowledge : 1;
        UINT64 CommandCompletedAcknowledge : 1;
    };
} UCSI_ACK_CC_CI_COMMAND, *PUCSI_ACK_CC_CI_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorChangeAcknowledge

CommandCompletedAcknowledge

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

<b>Requirement</b>	<b>Value</b>
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_ALTERNATE\_MODE structure (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_ALTERNATE\_MODES command. See Table 4-26 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_ALTERNATE_MODE {
    UINT16 SVID;
    union {
        struct {
            UINT16 ModeL;
            UINT16 ModeH;
        };
        UINT32 Mode;
    };
} UCSI_ALTERNATE_MODE, *PUCSI_ALTERNATE_MODE;
```

## Members

SVID

ModeL

ModeH

Mode

## Requirements

  Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_BATTERY\_CHARGING\_STATUS enumeration (ucmucsispes.h)

Article02/22/2024

See Table 4-42, Offset 64 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_BATTERY_CHARGING_STATUS {
    UcsiBatteryChargingNotCharging,
    UcsiBatteryChargingNominal,
    UcsiBatteryChargingSlowCharging,
    UcsiBatteryChargingTrickleCharging
} UCSI_BATTERY_CHARGING_STATUS;
```

## Constants

[Expand table](#)

UcsiBatteryChargingNotCharging

UcsiBatteryChargingNominal

UcsiBatteryChargingSlowCharging

UcsiBatteryChargingTrickleCharging

## Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_BM\_POWER\_SOURCE union (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_CAPABILITY command. See Bit 15:8 in Table 4-14 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_BM_POWER_SOURCE {
    UINT8 AsUInt8;
    struct {
        UINT8 AcSupply : 1;
        UINT8 Other : 1;
        UINT8 UsesVBus : 1;
    };
} UCSI_BM_POWER_SOURCE, *PUCSI_BM_POWER_SOURCE;
```

## Members

AsUInt8

AcSupply

Other

UsesVBus

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_CCI union (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_CONNECTOR\_CAPABILITY command. See Table 4-16 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_CCI {
    UINT32 AsUInt32;
    struct {
        UINT32 ConnectorChangeIndicator : 7;
        UINT32 DataLength : 8;
        UINT32 NotSupportedIndicator : 1;
        UINT32 CancelCompletedIndicator : 1;
        UINT32 ResetCompletedIndicator : 1;
        UINT32 BusyIndicator : 1;
        UINT32 AcknowledgeCommandIndicator : 1;
        UINT32 ErrorIndicator : 1;
        UINT32 CommandCompletedIndicator : 1;
    };
} UCSI_CCI, *PUCSI_CCI;
```

## Members

AsUInt32

ConnectorChangeIndicator

DataLength

NotSupportedIndicator

CancelCompletedIndicator

ResetCompletedIndicator

BusyIndicator

AcknowledgeCommandIndicator

ErrorIndicator

# Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_CMD\_SUCCEEDED function (ucmucsispes.h)

Article02/22/2024

On successful completion of a UCSI command the PPM firmware fills the CCI Data Structure provided by the client driver.

## Syntax

C++

```
BOOLEAN UCSI_CMD_SUCCEEDED(  
    UCSI_CCI Cci  
) ;
```

## Parameters

Cci

A [UCSI\\_CCI](#) structure.

## Return value

This function returns BOOLEAN.

## Requirements

  Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_COMMAND enumeration (ucmucsispes.h)

Article 06/03/2021

See Table 4-51: Command Code, in [UCSI spec version 1.1 ↗](#).

## Syntax

C++

```
typedef enum _UCSI_COMMAND {
    UcsiCommandPpmReset,
    UcsiCommandCancel,
    UcsiCommandConnectorReset,
    UcsiCommandAckCcCi,
    UcsiCommandSetNotificationEnable,
    UcsiCommandGetCapability,
    UcsiCommandGetConnectorCapability,
    UcsiCommandSetUom,
    UcsiCommandSetUor,
    UcsiCommandSetPdm,
    UcsiCommandSetPdr,
    UcsiCommandGetAlternateModes,
    UcsiCommandGetCamSupported,
    UcsiCommandGetCurrentCam,
    UcsiCommandSetNewCam,
    UcsiCommandGetPdos,
    UcsiCommandGetCableProperty,
    UcsiCommandGetConnectorStatus,
    UcsiCommandGetErrorStatus,
    UcsiCommandSetPowerLevel,
    UcsiCommandMax
} UCSI_COMMAND;
```

## Constants

`UcsiCommandPpmReset`

See PPM\_RESET described in section 4.5.1 of the specification.

`UcsiCommandCancel`

See CANCEL described in section 4.5.2 of the specification.

**UcsiCommandConnectorReset**

See CONNECTOR\_RESET described in section 4.5.3 of the specification.

**UcsiCommandAckCcCi**

See ACK\_CC\_CI described in section 4.5.4 of the specification.

**UcsiCommandSetNotificationEnable**

See SET\_NOTIFICATION\_ENABLE described in section 4.5.5 of the specification.

**UcsiCommandGetCapability**

See GET\_CAPABILITY described in section 4.5.6 of the specification.

**UcsiCommandGetConnectorCapability**

See GET\_CONNECTOR\_CAPABILITY described in section 4.5.7 of the specification.

**UcsiCommandSetUom**

See SET\_CCOM described in section 4.5.8 of the specification.

**UcsiCommandSetUor**

See SET\_UOR described in section 4.5.9 of the specification.

**UcsiCommandSetPdm**

See SET\_PDM described in section 4.5.10 of the specification.

**UcsiCommandSetPdr**

See SET\_PDR described in section 4.5.10 of the specification.

**UcsiCommandGetAlternateModes**

See GET\_ALTERNATE\_MODES described in section 4.5.11 of the specification.

**UcsiCommandGetCamSupported**

See GET\_CAM\_SUPPORTED described in section 4.5.12 of the specification.

**UcsiCommandGetCurrentCam**

See GET\_CURRENT\_CAM described in section 4.5.13 of the specification.

**UcsiCommandSetNewCam**

See SET\_NEW\_CAM described in section 4.5.14 of the specification.

**UcsiCommandGetPdos**

See GET\_PDOS described in section 4.5.15 of the specification.

**UcsiCommandGetCableProperty**

See GET\_CABLE\_PROPERTY described in section 4.5.16 of the specification.

**UcsiCommandGetConnectorStatus**

See GET\_CONNECTOR\_STATUS described in section 4.5.17 of the specification.

**UcsiCommandGetErrorStatus**

See GET\_ERROR\_STATUS described in section 4.5.18 of the specification.

**UcsiCommandSetPowerLevel**

See SET\_POWER\_LEVEL described in section 4.5.19 of the specification.

**UcsiCommandMax**

Reserved.

# Requirements

Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_CONNECTOR\_PARTNER\_FLAGS enumeration (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 21 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_CONNECTOR_PARTNER_FLAGS {
    UcsiConnectorPartnerFlagUsb,
    UcsiConnectorPartnerFlagAlternateMode
} UCSI_CONNECTOR_PARTNER_FLAGS;
```

## Constants

[ ] Expand table

UcsiConnectorPartnerFlagUsb

UcsiConnectorPartnerFlagAlternateMode

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_CONNECTOR\_PARTNER\_TYPE enumeration (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 29 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_CONNECTOR_PARTNER_TYPE {  
    UcsiConnectorPartnerTypeDfp,  
    UcsiConnectorPartnerTypeUfp,  
    UcsiConnectorPartnerTypePoweredCableNoUfp,  
    UcsiConnectorPartnerTypePoweredCableWithUfp,  
    UcsiConnectorPartnerTypeDebugAccessory,  
    UcsiConnectorPartnerTypeAudioAccessory  
} UCSI_CONNECTOR_PARTNER_TYPE;
```

## Constants

[+] Expand table

UcsiConnectorPartnerTypeDfp
UcsiConnectorPartnerTypeUfp
UcsiConnectorPartnerTypePoweredCableNoUfp
UcsiConnectorPartnerTypePoweredCableWithUfp
UcsiConnectorPartnerTypeDebugAccessory
UcsiConnectorPartnerTypeAudioAccessory

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_CONNECTOR\_RESET\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Used in the CONNECTOR\_RESET command. See Table 4-5 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_CONNECTOR_RESET_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 HardReset : 1;
    };
} UCSI_CONNECTOR_RESET_COMMAND, *PUCSI_CONNECTOR_RESET_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

HardReset

## Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_CONTROL union (ucmucsispec.h)

Article 02/22/2024

Used in the SET\_NOTIFICATION\_ENABLE command. See Table 4-9 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_CONTROL {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 CommandSpecific : 48;
    };
    UCSI_CONNECTOR_RESET_COMMAND ConnectorReset;
    UCSI_ACK_CC_CI_COMMAND AckCcCi;
    UCSI_SET_NOTIFICATION_ENABLE_COMMAND SetNotificationEnable;
    UCSI_GET_CONNECTOR_CAPABILITY_COMMAND GetConnectorCapability;
    UCSI_SET_UOM_COMMAND SetUom;
    UCSI_SET_UOR_COMMAND SetUor;
    UCSI_SET_PDM_COMMAND SetPdm;
    UCSI_SET_PDR_COMMAND SetPdr;
    UCSI_GET_ALTERNATE_MODES_COMMAND GetAlternateModes;
    UCSI_GET_CAM_SUPPORTED_COMMAND GetCamSupported;
    UCSI_GET_CURRENT_CAM_COMMAND GetCurrentCam;
    UCSI_SET_NEW_CAM_COMMAND SetNewCam;
    UCSI_GET_PDOS_COMMAND GetPdos;
    UCSI_GET_CABLE_PROPERTY_COMMAND GetCableProperty;
    UCSI_GET_CONNECTOR_STATUS_COMMAND GetConnectorStatus;
    UCSI_GET_ERROR_STATUS_COMMAND GetErrorStatus;
    UCSI_SET_POWER_LEVEL_COMMAND SetPowerLevel;
} UCSI_CONTROL, *PUCSI_CONTROL;
```

## Members

AsUInt64

Command

DataLength

CommandSpecific

`ConnectorReset`

`AckCcCi`

`SetNotificationEnable`

`GetConnectorCapability`

`SetUom`

`SetUor`

`SetPdm`

`SetPdr`

`GetAlternateModes`

`GetCamSupported`

`GetCurrentCam`

`SetNewCam`

`GetPdos`

`GetCableProperty`

`GetConnectorStatus`

`GetErrorStatus`

`SetPowerLevel`

## Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	<code>ucmucsispec.h</code> (include <code>UcmUcsiCx.h</code> )

# UCSI\_DATA\_BLOCK structure (ucmucsispes.h)

Article 02/22/2024

The data structures for memory locations. See Section 3 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_DATA_BLOCK {
    UCSI_VERSION      UcsiVersion;
    UCSI_CCI         CCI;
    UCSI_CONTROL     Control;
    UCSI_MESSAGE_IN   MessageIn;
    UCSI_MESSAGE_OUT  MessageOut;
} UCSI_DATA_BLOCK, *PUCSI_DATA_BLOCK;
```

## Members

UcsiVersion

CCI

Control

MessageIn

MessageOut

## Requirements

[] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_ALTERNATE\_MODES\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the GET\_ALTERNATE\_MODES command. See Table 4-24 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_ALTERNATE_MODES_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 Recipient : 3;
        UINT64 ConnectorNumber : 7;
        UINT64 AlternateModeOffset : 8;
        UINT64 NumberOfAlternateModes : 2;
    };
} UCSI_GET_ALTERNATE_MODES_COMMAND, *PUCSI_GET_ALTERNATE_MODES_COMMAND;
```

## Members

AsUInt64

Command

DataLength

Recipient

ConnectorNumber

AlternateModeOffset

NumberOfAlternateModes

## Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsisp.h (include UcmUcsiCx.h)

# UCSI\_GET\_ALTERNATE\_MODES\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_ALTERNATE\_MODES command. See Table 4-24 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_ALTERNATE_MODES_IN {
    UCSI_ALTERNATE_MODE AlternateModes[2];
} UCSI_GET_ALTERNATE_MODES_IN, *PUCSI_GET_ALTERNATE_MODES_IN;
```

## Members

AlternateModes[2]

## Requirements

[] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_ALTERNATE\_MODES\_RECIPIENT enumeration (ucmucsispec.h)

Article 02/22/2024

Used in the GET\_ALTERNATE\_MODES command. See Table 4-24, Offset 16 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_GET_ALTERNATE_MODES_RECIPIENT {
    UcsiGetAlternateModesRecipientConnector,
    UcsiGetAlternateModesRecipientSop,
    UcsiGetAlternateModesRecipientSopP,
    UcsiGetAlternateModesRecipientSopPP
} UCSI_GET_ALTERNATE_MODES_RECIPIENT;
```

## Constants

[ ] Expand table

UcsiGetAlternateModesRecipientConnector
UcsiGetAlternateModesRecipientSop
UcsiGetAlternateModesRecipientSopP
UcsiGetAlternateModesRecipientSopPP

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

<b>Requirement</b>	<b>Value</b>
Header	ucmucsispec.h (include UcmUcsiCx.h)

# **UCSI\_GET\_CABLE\_PROPERTY\_COMMAND**

## **Union (ucmucsispes.h)**

Article02/22/2024

Used in the GET\_CABLE\_PROPERTY command. See Table 4-37 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_CABLE_PROPERTY_COMMAND {  
    UINT64 AsUInt64;  
    struct {  
        UINT64 Command : 8;  
        UINT64 DataLength : 8;  
        UINT64 ConnectorNumber : 7;  
    };  
} UCSI_GET_CABLE_PROPERTY_COMMAND, *PUCSI_GET_CABLE_PROPERTY_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

## Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_CABLE\_PROPERTY\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CABLE\_PROPERTY command. See Table 4-39 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_CABLE_PROPERTY_IN {
    union {
        UINT16 AsUInt16;
        struct {
            UINT16 SpeedExponent : 2;
            UINT16 Mantissa : 14;
        };
    } bmSpeedSupported;
    UINT8 bCurrentCapability;
    UINT16 VBusInCable : 1;
    UINT16 CableType : 1;
    UINT16 Directionality : 1;
    UINT16 PlugEndType : 2;
    UINT16 ModeSupport : 1;
    UINT16 Latency : 4;
} UCSI_GET_CABLE_PROPERTY_IN, *PUCSI_GET_CABLE_PROPERTY_IN;
```

## Members

bmSpeedSupported

bmSpeedSupported.AsUInt16

bmSpeedSupported.SpeedExponent

bmSpeedSupported.Mantissa

bCurrentCapability

VBusInCable

CableType

Directionality

PlugEndType

ModeSupport

Latency

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_GET\_CAM\_SUPPORTED\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CAM\_SUPPORTED command. See Table 4-27 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_CAM_SUPPORTED_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
    };
} UCSI_GET_CAM_SUPPORTED_COMMAND, *PUCSI_GET_CAM_SUPPORTED_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

## Requirements

[] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_CAM\_SUPPORTED\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CAM\_SUPPORTED command. See Table 4-27 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_CAM_SUPPORTED_IN {
    UINT8 bmAlternateModeSupported[16];
} UCSI_GET_CAM_SUPPORTED_IN, *PUCSI_GET_CAM_SUPPORTED_IN;
```

## Members

bmAlternateModeSupported[16]

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_CAPABILITY\_IN structure (ucmucsispec.h)

Article02/22/2024

Used in the GET\_CAPABILITY command. See Table 4-13 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_CAPABILITY_IN {
    union {
        UINT32 AsUInt32;
        struct {
            UINT32 DisabledStateSupport : 1;
            UINT32 BatteryCharging : 1;
            UINT32 UsbPowerDelivery : 1;
            UINT32 UsbTypeCCurrent : 1;
            UINT32 bmPowerSource : 8;
        };
    } bmAttributes;
    union {
        UINT8 bNumConnectors : 7;
    };
    union {
        struct {
            UINT32 SetUomSupported : 1;
            UINT32 SetPdmSupported : 1;
            UINT32 AlternateModeDetailsAvailable : 1;
            UINT32 AlternateModeOverrideSupported : 1;
            UINT32 PdoDetailsAvailable : 1;
            UINT32 CableDetailsAvailable : 1;
            UINT32 ExternalSupplyNotificationSupported : 1;
            UINT32 PdResetNotificationSupported : 1;
        } bmOptionalFeatures;
        struct {
            UINT32 OptionalFeatures : 24;
            UINT32 bNumAltModes : 8;
        };
    };
    UINT16 bcdBcVersion;
    UINT16 bcdPdVersion;
    UINT16 bcdUsbTypeCVersion;
} UCSI_GET_CAPABILITY_IN, *PUCSI_GET_CAPABILITY_IN;
```

## Members

```
bmAttributes
```

```
bmAttributes.AsUInt32
```

```
bmAttributes.DisabledStateSupport
```

```
bmAttributes.BatteryCharging
```

```
bmAttributes.UsbPowerDelivery
```

```
bmAttributes.UsbTypeCCurrent
```

```
bmAttributes.bmPowerSource
```

```
bNumConnectors
```

```
bmOptionalFeatures
```

```
bmOptionalFeatures.SetUomSupported
```

```
bmOptionalFeatures.SetPdmSupported
```

```
bmOptionalFeatures.AlternateModeDetailsAvailable
```

```
bmOptionalFeatures.AlternateModeOverrideSupported
```

```
bmOptionalFeatures.PdoDetailsAvailable
```

```
bmOptionalFeatures.CableDetailsAvailable
```

```
bmOptionalFeatures.ExternalSupplyNotificationSupported
```

```
bmOptionalFeatures.PdResetNotificationSupported
```

```
OptionalFeatures
```

```
bNumAltModes
```

```
bcdBcVersion
```

```
bcdPdVersion
```

```
bcdUsbTypeCVersion
```

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsisp.h (include UcmUcsiCx.h)

# UCSI\_GET\_CONNECTOR\_CAPABILITY\_COMMAND union (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_CONNECTOR\_CAPABILITY command. See Table 4-15 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_CONNECTOR_CAPABILITY_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
    };
} UCSI_GET_CONNECTOR_CAPABILITY_COMMAND,
*PUCSI_GET_CONNECTOR_CAPABILITY_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

## Requirements

  Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_GET\_CONNECTOR\_CAPABILITY\_IN structure (ucmucsispes.h)

Article01/06/2022

Used in the GET\_CONNECTOR\_CAPABILITY command.

## Syntax

C++

```
typedef struct _UCSI_GET_CONNECTOR_CAPABILITY_IN {
    union {
        UINT8 AsUInt8;
        struct {
            UINT8 DfpOnly : 1;
            UINT8 UfpOnly : 1;
            UINT8 Drp : 1;
            UINT8 AudioAccessoryMode : 1;
            UINT8 DebugAccessoryMode : 1;
            UINT8 Usb2 : 1;
            UINT8 Usb3 : 1;
            UINT8 AlternateMode : 1;
        };
    } OperationMode;
    UINT8 Provider : 1;
    UINT8 Consumer : 1;
    UINT8 SwapToDfp : 1;
    UINT8 SwapToUfp : 1;
    UINT8 SwapToSrc : 1;
    UINT8 SwapToSnk : 1;
} UCSI_GET_CONNECTOR_CAPABILITY_IN, *PUCSI_GET_CONNECTOR_CAPABILITY_IN;
```

## Members

OperationMode

This field indicates the mode that the connector can support.

OperationMode.AsUInt8

For internal use.

OperationMode.DfpOnly

Indicates that the connector supports only DFP mode.

`OperationMode.UfpOnly`

Indicates that the connector supports only UFP mode.

`OperationMode.Drp`

Indicates that the connector supports DRP mode.

`OperationMode.AudioAccessoryMode`

Indicates that the connector supports audio accessory mode.

`OperationMode.DebugAccessoryMode`

Indicates that the connector supports debug accessory mode.

`OperationMode.Usb2`

Indicates that the connector supports USB2 mode.

`OperationMode.Usb3`

Indicates that the connector supports USB3 mode.

`OperationMode.AlternateMode`

Indicates that the connector supports an alternate mode.

`Provider`

Indicates that the connector is capable of providing power.

`Consumer`

Indicates that the connector is capable of consuming power.

`SwapToDfp`

Indicates that the connector is capable of accepting swap to DFP.

`SwapToUfp`

Indicates that the connector is capable of accepting swap to UFP.

`SwapToSrc`

Indicates that the connector is capable of accepting swap to SRC.

## SwapToSnk

Indicates that the connector is capable of accepting swap to SNK.

## Remarks

See Table 4-17 in [UCSI spec version 1.2 ↗](#).

## Requirements

<b>Minimum KMDF version</b>	1.27
<b>Minimum UMDF version</b>	N/A
<b>Header</b>	ucmucsispec.h (include UcmUcsiCx.h)

## See also

- [UCSI\\_GET\\_CONNECTOR\\_CAPABILITY\\_COMMAND](#)
- [UCSI spec version 1.2 ↗](#)

# UCSI\_GET\_CONNECTOR\_STATUS\_COMM AND union (ucmucsisp.h)

Article02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-40 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_CONNECTOR_STATUS_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
    };
} UCSI_GET_CONNECTOR_STATUS_COMMAND, *PUCSI_GET_CONNECTOR_STATUS_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsisp.h (include UcmUcsiCx.h)

# UCSI\_GET\_CONNECTOR\_STATUS\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_CONNECTOR_STATUS_IN {
    union {
        UINT16 AsUInt16;
        struct {
            UINT16 ExternalSupplyChange : 1;
            UINT16 PowerOperationModeChange : 1;
            UINT16 SupportedProviderCapabilitiesChange : 1;
            UINT16 NegotiatedPowerLevelChange : 1;
            UINT16 PdResetComplete : 1;
            UINT16 SupportedCamChange : 1;
            UINT16 BatteryChargingStatusChange : 1;
            UINT16 ConnectorPartnerChange : 1;
            UINT16 PowerDirectionChange : 1;
            UINT16 ConnectChange : 1;
            UINT16 Error : 1;
        };
        } ConnectorStatusChange;
        UINT16 PowerOperationMode : 3;
        UINT16 ConnectStatus : 1;
        UINT16 PowerDirection : 1;
        UINT16 ConnectorPartnerFlags : 8;
        UINT16 ConnectorPartnerType : 3;
        UINT32 RequestDataObject;
    union {
        struct {
            UINT8 BatteryChargingStatus : 2;
            UINT8 PowerBudgetLimitedReason : 4;
        };
        struct {
            UINT8 PowerBudgetLowered : 1;
            UINT8 ReachingPowerBudgetLimit : 1;
        } bmPowerBudgetLimitedReason;
    };
} UCSI_GET_CONNECTOR_STATUS_IN, *PUCSI_GET_CONNECTOR_STATUS_IN;
```

# Members

ConnectorStatusChange

ConnectorStatusChange.AsUInt16

ConnectorStatusChange.ExternalSupplyChange

ConnectorStatusChange.PowerOperationModeChange

ConnectorStatusChange.SupportedProviderCapabilitiesChange

ConnectorStatusChange.NegotiatedPowerLevelChange

ConnectorStatusChange.PdResetComplete

ConnectorStatusChange.SupportedCamChange

ConnectorStatusChange.BatteryChargingStatusChange

ConnectorStatusChange.ConnectorPartnerChange

ConnectorStatusChange.PowerDirectionChange

ConnectorStatusChange.ConnectChange

ConnectorStatusChange.Error

PowerOperationMode

ConnectStatus

PowerDirection

ConnectorPartnerFlags

ConnectorPartnerType

RequestDataObject

BatteryChargingStatus

PowerBudgetLimitedReason

bmPowerBudgetLimitedReason

bmPowerBudgetLimitedReason.PowerBudgetLowered

bmPowerBudgetLimitedReason.ReachingPowerBudgetLimit

# Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_GET\_CURRENT\_CAM\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CURRENT\_CAM command. See Table 4-29 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_CURRENT_CAM_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
    };
} UCSI_GET_CURRENT_CAM_COMMAND, *PUCSI_GET_CURRENT_CAM_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

## Requirements

  Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_CURRENT\_CAM\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CURRENT\_CAM command. See Table 4-31 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_CURRENT_CAM_IN {  
    UINT8 CurrentAlternateMode;  
} UCSI_GET_CURRENT_CAM_IN, *PUCSI_GET_CURRENT_CAM_IN;
```

## Members

CurrentAlternateMode

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# **UCSI\_GET\_ERROR\_STATUS\_COMMAND**

## **union (ucmucsispes.h)**

Article02/22/2024

Used in the GET\_ERROR\_STATUS command. See Table 4-45 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_ERROR_STATUS_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
    };
} UCSI_GET_ERROR_STATUS_COMMAND, *PUCSI_GET_ERROR_STATUS_COMMAND;
```

## Members

AsUInt64

Command

DataLength

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_ERROR\_STATUS\_IN structure (ucmucsispec.h)

Article02/22/2024

Used in the GET\_ERROR\_STATUS command. See Table 4-47 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_ERROR_STATUS_IN {
    union {
        UINT16 AsUInt16;
        struct {
            UINT16 UnrecognizedCommandError : 1;
            UINT16 NonExistentConnectorNumberError : 1;
            UINT16 InvalidCommandParametersError : 1;
            UINT16 IncompatibleConnectorPartnerError : 1;
            UINT16 CcCommunicationError : 1;
            UINT16 CommandFailureDueToDeadBattery : 1;
            UINT16 ContractNegotiationFailure : 1;
        };
    } ErrorInformation;
    UINT8 VendorDefined[14];
} UCSI_GET_ERROR_STATUS_IN, *PUCSI_GET_ERROR_STATUS_IN;
```

## Members

ErrorInformation

ErrorInformation.AsUInt16

ErrorInformation.UnrecognizedCommandError

ErrorInformation.NonExistentConnectorNumberError

ErrorInformation.InvalidCommandParametersError

ErrorInformation.IncompatibleConnectorPartnerError

ErrorInformation.CcCommunicationError

ErrorInformation.CommandFailureDueToDeadBattery

ErrorInformation.ContractNegotiationFailure

VendorDefined[14]

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_GET\_PDOS\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the GET\_PDOS command. See Table 4-34 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_GET_PDOS_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 PartnerPdo : 1;
        UINT64 PdoOffset : 8;
        UINT64 NumberOfPdos : 2;
        UINT64 SourceOrSinkPdos : 1;
        UINT64 SourceCapabilitiesType : 2;
    };
} UCSI_GET_PDOS_COMMAND, *PUCSI_GET_PDOS_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

PartnerPdo

PdoOffset

NumberOfPdos

SourceOrSinkPdos

SourceCapabilitiesType

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_GET\_PDOS\_IN structure (ucmucsispes.h)

Article02/22/2024

Used in the GET\_PDOS command. See Table 4-36 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef struct _UCSI_GET_PDOS_IN {
    UINT32 Pdos[4];
} UCSI_GET_PDOS_IN, *PUCSI_GET_PDOS_IN;
```

## Members

Pdos[4]

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# **UCSI\_GET\_PDOS\_SOURCE\_CAPABILITIES\_TYPE enumeration (ucmucsispes.h)**

Article 02/22/2024

Used in the GET\_PDOS command. See Table 4-34, Offset 35 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_GET_PDOS_SOURCE_CAPABILITIES_TYPE {  
    UcsiGetPdosCurrentSourceCapabilities,  
    UcsiGetPdosAdvertisedSourceCapabilities,  
    UcsiGetPdosMaxSourceCapabilities  
} UCSI_GET_PDOS_SOURCE_CAPABILITIES_TYPE;
```

## Constants

[+] Expand table

UcsiGetPdosCurrentSourceCapabilities
UcsiGetPdosAdvertisedSourceCapabilities
UcsiGetPdosMaxSourceCapabilities

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_GET\_PDOS\_TYPE enumeration (ucmucsispes.h)

Article02/22/2024

Used in the GET\_PDOS command. See Table 4-34, Offset 34 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_GET_PDOS_TYPE {  
    UcsiGetPdosTypeSink,  
    UcsiGetPdosTypeSource  
} UCSI_GET_PDOS_TYPE;
```

## Constants

[] [Expand table](#)

UcsiGetPdosTypeSink
UcsiGetPdosTypeSource

## Requirements

[] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_MESSAGE\_IN union (ucmucsispec.h)

Article 02/22/2024

The MESSAGE IN data structure. See Section 3.4 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_MESSAGE_IN {
    UINT8                                AsBuffer[UCSI_MAX_DATA_LENGTH];
    UCSI_GET_CAPABILITY_IN                 Capability;
    UCSI_GET_CONNECTOR_CAPABILITY_IN       ConnectorCapability;
    UCSI_GET_ALTERNATE_MODES_IN           AlternateModes;
    UCSI_GET_CAM_SUPPORTED_IN             CamSupported;
    UCSI_GET_CURRENT_CAM_IN               CurrentCam;
    UCSI_GET_PDOS_IN                     Pdos;
    UCSI_GET_CABLE_PROPERTY_IN            CableProperty;
    UCSI_GET_CONNECTOR_STATUS_IN          ConnectorStatus;
    UCSI_GET_ERROR_STATUS_IN              ErrorStatus;
} UCSI_MESSAGE_IN, *PUCSI_MESSAGE_IN;
```

## Members

AsBuffer[UCSI\_MAX\_DATA\_LENGTH]

Capability

ConnectorCapability

AlternateModes

CamSupported

CurrentCam

Pdos

CableProperty

ConnectorStatus

# Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_MESSAGE\_OUT union (ucmucsispes.h)

Article02/22/2024

The MESSAGE OUT data structure. See Section 3.5 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_MESSAGE_OUT {
    UINT8 AsBuffer[UCSI_MAX_DATA_LENGTH];
} UCSI_MESSAGE_OUT, *PUCSI_MESSAGE_OUT;
```

## Members

AsBuffer[UCSI\_MAX\_DATA\_LENGTH]

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_POWER\_DIRECTION enumeration (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 20 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_POWER_DIRECTION {
    UcsiPowerDirectionConsumer,
    UcsiPowerDirectionProvider
} UCSI_POWER_DIRECTION;
```

## Constants

[] Expand table

UcsiPowerDirectionConsumer
UcsiPowerDirectionProvider

## Requirements

[] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_POWER\_DIRECTION\_MODE enumeration (ucmucsispes.h)

Article02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 20 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_POWER_DIRECTION_MODE {  
    UcsiPowerDirectionModeProvider,  
    UcsiPowerDirectionModeConsumer,  
    UcsiPowerDirectionModeEither  
} UCSI_POWER_DIRECTION_MODE;
```

## Constants

[ ] Expand table

UcsiPowerDirectionModeProvider
UcsiPowerDirectionModeConsumer
UcsiPowerDirectionModeEither

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_POWER\_DIRECTION\_ROLE enumeration (ucmucsispes.h)

Article 05/22/2024

Used in the SET\_PDR command. The SET\_PDR command is used to set the power direction dictated by the OS Policy Manager (OPM), for the current connection.

## Syntax

C++

```
typedef enum _UCSI_POWER_DIRECTION_ROLE {
    UcsiPowerDirectionRoleProvider = 0x1,
    UcsiPowerDirectionRoleConsumer = 0x2,
    UcsiPowerDirectionRoleAcceptSwap = 0x4,
    UcsiPowerDirectionRoleProviderAcceptSwap = 0x5,
    UcsiPowerDirectionRoleConsumerAcceptSwap = 0x6
} UCSI_POWER_DIRECTION_ROLE;
```

## Constants

[+] Expand table

<b>UcsiPowerDirectionRoleProvider</b> Value: 0x1 The connector initiates swap to source, if not already operating as source.
<b>UcsiPowerDirectionRoleConsumer</b> Value: 0x2 The connector initiates swap to sink, if not already operating as sink.
<b>UcsiPowerDirectionRoleAcceptSwap</b> Value: 0x4 The connector accepts power direction swap requests from the port partner. If this bit is cleared, the connector rejects power direction swap requests from the port partner.
<b>UcsiPowerDirectionRoleProviderAcceptSwap</b> Value: 0x5 This field combines the <i>UcsiPowerDirectionRoleProvider</i> and <i>UcsiPowerDirectionRoleAcceptSwap</i> values.

#### `UcsiPowerDirectionRoleConsumerAcceptSwap`

Value: `0x6`

This field combines the `UcsiPowerDirectionRoleConsumer` and `UcsiPowerDirectionRoleAcceptSwap` values.

## Remarks

For more information, see section 4.5.10 in the [UCSI spec version 1.2](#).

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	<code>ucmucsisp.h</code> (include <code>UcmUcsiCx.h</code> )

## See also

- [UCSI spec version 1.2](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

# UCSI\_POWER\_OPERATION\_MODE enumeration (ucmucsispes.h)

Article 02/22/2024

Used in the GET\_CONNECTOR\_STATUS command. See Table 4-42, Offset 16 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_POWER_OPERATION_MODE {  
    UcsiPowerOperationModeNoConsumer,  
    UcsiPowerOperationModeDefaultUsb,  
    UcsiPowerOperationModeBc,  
    UcsiPowerOperationModePd,  
    UcsiPowerOperationModeTypeC1500,  
    UcsiPowerOperationModeTypeC3000  
} UCSI_POWER_OPERATION_MODE;
```

## Constants

[+] Expand table

UcsiPowerOperationModeNoConsumer
UcsiPowerOperationModeDefaultUsb
UcsiPowerOperationModeBc
UcsiPowerOperationModePd
UcsiPowerOperationModeTypeC1500
UcsiPowerOperationModeTypeC3000

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_NEW\_CAM\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the SET\_NEW\_CAM command. See Table 4-32 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_NEW_CAM_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 EnterOrExit : 1;
        UINT64 NewCam : 8;
        UINT64 AmSpecific : 32;
    };
} UCSI_SET_NEW_CAM_COMMAND, *PUCSI_SET_NEW_CAM_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

EnterOrExit

NewCam

AmSpecific

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_NOTIFICATION\_ENABLE\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the SET\_NOTIFICATION\_ENABLE command. See Table 4-9 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_NOTIFICATION_ENABLE_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT8 Command;
        UINT8 DataLength;
        union {
            UINT16 NotificationEnable;
            struct {
                UINT16 CommandCompleteNotificationEnable : 1;
                UINT16 ExternalSupplyChangeNotificationEnable : 1;
                UINT16 PowerOperationModeChangeNotificationEnable : 1;
                UINT16 SupportedProviderCapabilitiesChangeNotificationEnable : 1;
                UINT16 NegotiatedPowerLevelChangeNotificationEnable : 1;
                UINT16 PdResetNotificationEnable : 1;
                UINT16 SupportedCamChangeNotificationEnable : 1;
                UINT16 BatteryChargingStatusChangeNotificationEnable : 1;
                UINT16 DataRoleSwapCompletedNotificationEnable : 1;
                UINT16 PowerRoleSwapCompletedNotificationEnable : 1;
                UINT16 ConnectChangeNotificationEnable : 1;
                UINT16 ErrorNotificationEnable : 1;
            };
        };
    };
} UCSI_SET_NOTIFICATION_ENABLE_COMMAND,
*PUCSI_SET_NOTIFICATION_ENABLE_COMMAND;
```

## Members

AsUInt64

Command

DataLength

`NotificationEnable`

`CommandCompleteNotificationEnable`

`ExternalSupplyChangeNotificationEnable`

`PowerOperationModeChangeNotificationEnable`

`SupportedProviderCapabilitiesChangeNotificationEnable`

`NegotiatedPowerLevelChangeNotificationEnable`

`PdResetNotificationEnable`

`SupportedCamChangeNotificationEnable`

`BatteryChargingStatusChangeNotificationEnable`

`DataRoleSwapCompletedNotificationEnable`

`PowerRoleSwapCompletedNotificationEnable`

`ConnectChangeNotificationEnable`

`ErrorNotificationEnable`

## Requirements

[\[\] Expand table](#)

Requirement	Value
<b>Minimum KMDF version</b>	1.27
<b>Minimum UMDF version</b>	N/A
<b>Header</b>	<code>ucmucsisp.h</code> (include <code>UcmUcsiCx.h</code> )

# UCSI\_SET\_PDM\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Obsolete.

## Syntax

C++

```
typedef union _UCSI_SET_PDM_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 PowerDirectionMode : 3;
    };
} UCSI_SET_PDM_COMMAND, *PUCSI_SET_PDM_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

PowerDirectionMode

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

<b>Requirement</b>	<b>Value</b>
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_PDR\_COMMAND union (ucmucsispes.h)

Article02/22/2024

Used in the SET\_PDR command. See Table 4-22 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_PDR_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 PowerDirectionRole : 3;
    };
} UCSI_SET_PDR_COMMAND, *PUCSI_SET_PDR_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

PowerDirectionRole

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_POWER\_LEVEL\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the SET\_POWER\_LEVEL command. See Table 4-48 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_POWER_LEVEL_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 SourceOrSink : 1;
        UINT64 UsbPdMaxPowerIn500mW : 8;
        UINT64 UsbTypeCCurrent : 2;
    };
} UCSI_SET_POWER_LEVEL_COMMAND, *PUCSI_SET_POWER_LEVEL_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

SourceOrSink

UsbPdMaxPowerIn500mW

UsbTypeCCurrent

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_UOM\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the SET\_CCOM command. See Table 4-18 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_UOM_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 UsbOperationMode : 3;
    };
} UCSI_SET_UOM_COMMAND, *PUCSI_SET_UOM_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

UsbOperationMode

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_SET\_UOR\_COMMAND union (ucmucsispec.h)

Article02/22/2024

Used in the SET\_UOR command. See Table 4-20 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_SET_UOR_COMMAND {
    UINT64 AsUInt64;
    struct {
        UINT64 Command : 8;
        UINT64 DataLength : 8;
        UINT64 ConnectorNumber : 7;
        UINT64 UsbOperationRole : 3;
    };
} UCSI_SET_UOR_COMMAND, *PUCSI_SET_UOR_COMMAND;
```

## Members

AsUInt64

Command

DataLength

ConnectorNumber

UsbOperationRole

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	ucmucsispec.h (include UcmUcsiCx.h)

# UCSI\_USB\_OPERATION\_MODE enumeration (ucmucsispes.h)

Article 02/22/2024

Used in the SET\_UOR command. See Table 4-18, Offset 23 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef enum _UCSI_USB_OPERATION_MODE {
    UcsiUsbOperationModeDfp,
    UcsiUsbOperationModeUfp,
    UcsiUsbOperationModeDrp
} UCSI_USB_OPERATION_MODE;
```

## Constants

[+] Expand table

UcsiUsbOperationModeDfp
UcsiUsbOperationModeUfp
UcsiUsbOperationModeDrp

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispes.h (include UcmUcsiCx.h)

# UCSI\_USB\_OPERATION\_ROLE enumeration (ucmucsispes.h)

Article 05/22/2024

Used in the SET\_UOR command. The SET\_UOR command is used to set the USB operation role dictated by the OS Policy Manager (OPM), for the current connection.

## Syntax

C++

```
typedef enum _UCSI_USB_OPERATION_ROLE {
    UcsiUsbOperationRoleDfp = 0x1,
    UcsiUsbOperationRoleUfp = 0x2,
    UcsiUsbOperationRoleAcceptSwap = 0x4,
    UcsiUsbOperationRoleDfpAcceptSwap = 0x5,
    UcsiUsbOperationRoleUfpAcceptSwap = 0x6
} UCSI_USB_OPERATION_ROLE;
```

## Constants

[+] Expand table

<b>UcsiUsbOperationRoleDfp</b> Value: 0x1 The connector initiates swap to downstream-facing port (DFP), if not already operating in DFP mode.
<b>UcsiUsbOperationRoleUfp</b> Value: 0x2 The connector initiates swap to upstream-facing port (UFP), if not already operating in UFP mode.
<b>UcsiUsbOperationRoleAcceptSwap</b> Value: 0x4 The connector accepts USB operation role swap requests from the port partner. If this bit is cleared, connector rejects role swap requests from the port partner.
<b>UcsiUsbOperationRoleDfpAcceptSwap</b> Value: 0x5 This field combines the <i>UcsiUsbOperationRoleDfp</i> and <i>UcsiUsbOperationRoleAcceptSwap</i> values.

`UcsiUsbOperationRoleUfpAcceptSwap`

Value: `0x6`

This field combines the `UcsiUsbOperationRoleUfp` and `UcsiUsbOperationRoleAcceptSwap` values.

## Remarks

For more information, see section 4.5.9 in the [UCSI spec version 1.2](#).

## Requirements

[Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	<code>ucmucsispec.h</code> (include <code>UcmUcsiCx.h</code> )

## See also

- [UCSI spec version 1.2](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | Get help at Microsoft Q&A

# UCSI\_VERSION union (ucmucsispec.h)

Article02/22/2024

The VERSION data structure. See Section 3.1 in [UCSI spec version 1.2](#).

## Syntax

C++

```
typedef union _UCSI_VERSION {
    UINT16 AsUInt16;
    struct {
        UINT16 SubMinorVersion : 4;
        UINT16 MinorVersion : 4;
        UINT16 MajorVersion : 8;
    };
} UCSI_VERSION, *PUCSI_VERSION;
```

## Members

AsUInt16

SubMinorVersion

MinorVersion

MajorVersion

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	ucmucsispec.h (include UcmUcsiCx.h)

# ucxclass.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucxclass.h contains the following programming interfaces:

## Functions

### [UcxInitializeDeviceInit](#)

UcxInitializeDeviceInit initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

# UcxInitializeDeviceInit function (ucxclass.h)

Article02/22/2024

Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

## Syntax

C++

```
NTSTATUS UcxInitializeDeviceInit(  
    [in, out] PWDFDEVICE_INIT DeviceInit  
)
```

## Parameters

[in, out] DeviceInit

A pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure.

## Return value

(NTSTATUS) The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method may return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the host controller calls this method in its [EvtDriverDeviceAdd](#) implementation before it calls [WdfDeviceCreate](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Target Platform	Windows
Header	ucxclass.h
IRQL	PASSIVE_LEVEL

## See also

[WdfDeviceCreate](#)

# ucxcontroller.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucxcontroller.h contains the following programming interfaces:

## Functions

<a href="#">UCX_CONTROLLER_CONFIG_SET_ACPI_INFO</a>
Initializes a UCX_CONTROLLER_CONFIG structure with the specified values for the controller with ACPI as the parent.
<a href="#">UCX_CONTROLLER_CONFIG_SET_PCI_INFO</a>
Initializes a UCX_CONTROLLER_CONFIG structure with the specified values for the controller with PCI as the parent bus type.
<a href="#">UcxControllerCreate</a>
Creates a host controller object.
<a href="#">UcxControllerNeedsReset</a>
Initiates a non-Plug and Play (PnP) controller reset operation by queuing an event into the controller reset state machine.
<a href="#">UcxControllerNotifyTransportCharacteristicsChange</a>
Notifies UCX about a new port change event from host controller.
<a href="#">UcxControllerResetComplete</a>
Informs USB host controller extension (UCX) that the reset operation has completed.
<a href="#">UcxControllerSetFailed</a>
Informs USB Host Controller Extension (UCX) that the controller has encountered a critical failure.
<a href="#">UcxControllerSetIdStrings</a>
Updates the identifier strings of a controller after the controller has been initialized.

## [UcxIoDeviceControl](#)

Allows USB host controller extension (UCX) to handle an I/O control code (IOCTL) request from user mode.

# Callback functions

## [EVT\\_UCH\\_CONTROLLER\\_GET\\_CURRENT\\_FRAMENUMBER](#)

The client driver's implementation that UCX calls to retrieve the current 32-bit frame number.

## [EVT\\_UCH\\_CONTROLLER\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback to retrieves the system query performance counter (QPC) value synchronized with the frame and microframe.

## [EVT\\_UCH\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

UCX invokes this callback to retrieve the host controller characteristics.

## [EVT\\_UCH\\_CONTROLLER\\_QUERY\\_USB\\_CAPABILITY](#)

The client driver's implementation to determine if the controller supports a specific capability.

## [EVT\\_UCH\\_CONTROLLER\\_RESET](#)

The client driver's implementation that UCX calls to reset the controller.

## [EVT\\_UCH\\_CONTROLLER\\_SET\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#)

UCX invokes this callback function to specify its preference in transport characteristics for which the client driver must send notifications when changes occur.

## [EVT\\_UCH\\_CONTROLLER\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback function to the start time tracking functionality in the controller.

## [EVT\\_UCH\\_CONTROLLER\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

UCX invokes this callback function to the stop time tracking functionality in the controller.

## [EVT\\_UCH\\_CONTROLLER\\_USBDEVICE\\_ADD](#)

The client driver's implementation that UCX calls when a new USB device is detected.

# Structures

## [UCX\\_CONTROLLER\\_ACPI\\_INFORMATION](#)

This structure provides information about an advanced Configuration and power interface (ACPI) USB controller.

## [UCX\\_CONTROLLER\\_CONFIG](#)

This structure configuration data for a USB controller.

## [UCX\\_CONTROLLER\\_PCI\\_INFORMATION](#)

This structure provides information about a PCI USB controller.

## [UCX\\_CONTROLLER\\_RESET\\_COMPLETE\\_INFO](#)

Contains information about the operation to reset the controller. This is used by the client driver in its EVT\_UCX\_CONTROLLER\_RESET callback function.

## [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS](#)

Stores the transport characteristics at relevant points in time. This structure is used in the EVT\_UCX\_CONTROLLER\_GET\_TRANSPORT\_CHARACTERISTICS callback function.

## [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_FLAGS](#)

Defines flags for the transport characteristics changes. This structure is used in the EVT\_UCX\_CONTROLLER\_SET\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_NOTIFICATION callback function.

# Enumerations

## [UCX\\_CONTROLLER\\_PARENT\\_BUS\\_TYPE](#)

The UCX\_CONTROLLER\_PARENT\_BUS\_TYPE enumeration defines the parent bus type.

## [UCX\\_CONTROLLER\\_STATE](#)

This enumeration provides values to specify the UCX controller state after a reset.

# EVT\_UCX\_CONTROLLER\_GET\_CURRENT\_FRAMENUMBER callback function (ucxcontroller.h)

Article 10/21/2021

The client driver's implementation that UCX calls to retrieve the current 32-bit frame number.

## Syntax

C++

```
EVT_UCX_CONTROLLER_GET_CURRENT_FRAMENUMBER
EvtUcxControllerGetCurrentFramenumber;

NTSTATUS EvtUcxControllerGetCurrentFramenumber(
    [in] UCXCONTROLLER UcxController,
    [out] PULONG FrameNumber
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[out] FrameNumber

A pointer to the current 32-bit frame number.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers its *EVT\_UCX\_CONTROLLER\_GET\_CURRENT\_FRAMENUMBER* implementation with the USB host controller extension (UCX) by calling the [UcxControllerCreate](#) method.

## Examples

```
NTSTATUS
Controller_EvtControllerGetCurrentFrameNumber(
    UCXCONTROLLER    UcxController,
    PULONG           FrameNumber
)
{
    UNREFERENCED_PARAMETER(UcxController);

    //
    // TODO: Return the current 32-bit frame number.  Do not access the
    // controller registers if the controller is not in D0.
    //

    *FrameNumber = 0xFFFFFFFF;

    DbgTrace(TL_INFO, Controller,
    "Controller_EvtControllerGetCurrentFrameNumber");

    return STATUS_SUCCESS;
}
```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

# EVT\_UCX\_CONTROLLER\_GET\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC callback function (ucxcontroller.h)

Article02/22/2024

UCX invokes this callback to retrieves the system query performance counter (QPC) value synchronized with the frame and microframe.

## Syntax

C++

```
EVT_UCX_CONTROLLER_GET_FRAME_NUMBER_AND_QPC_FOR_TIME_SYNC  
EvtUcxControllerGetFrameNumberAndQpcForTimeSync;  
  
void EvtUcxControllerGetFrameNumberAndQpcForTimeSync(  
    [in] UCXCONTROLLER UcxController,  
    [in] WDFREQUEST WdfRequest,  
    [in] size_t OutputBufferLength,  
    [in] size_t InputBufferLength  
)  
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] WdfRequest

A framework request object that contains the request to get the synchronized frame and microframe number.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available. This value is the size of the [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available. This value is the size of the [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## Return value

None

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[IOCTL\\_USB\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#)

# EVT\_UCX\_CONTROLLER\_GET\_TRANSPORT\_CHARACTERISTICS callback function (ucxcontroller.h)

Article02/22/2024

UCX invokes this callback to retrieve the host controller characteristics.

## Syntax

C++

```
EVT_UCX_CONTROLLER_GET_TRANSPORT_CHARACTERISTICS  
EvtUcxControllerGetTransportCharacteristics;  
  
NTSTATUS EvtUcxControllerGetTransportCharacteristics(  
    [in] UCXCONTROLLER UcxController,  
    [out] PUCX_CONTROLLER_TRANSPORT_CHARACTERISTICS  
UcxControllerTransportCharacteristics  
)  
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[out] UcxControllerTransportCharacteristics

A pointer to a [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS](#) structure that the client driver for the host controller fills with transport characteristics.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers its implementation with the USB host controller extension (UCX) by calling the [UcxControllerCreate](#) method.

This callback function is optional. Whenever transport characteristics change, the client driver is responsible for notifying UCX that one of the characteristics have changed using a new function [UcxControllerNotifyTransportCharacteristicsChange](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

# EVT\_UCX\_CONTROLLER\_QUERY\_USB\_CA PABILITY callback function (ucxcontroller.h)

Article 10/21/2021

The client driver's implementation to determine if the controller supports a specific capability.

## Syntax

C++

```
EVT_UCX_CONTROLLER_QUERY_USB_CAPABILITY EvtUcxControllerQueryUsbCapability;

NTSTATUS EvtUcxControllerQueryUsbCapability(
    [in]          UCXCONTROLLER UcxController,
    [in]          PGUID CapabilityType,
    [in]          ULONG OutputBufferLength,
    [out, optional] PVOID OutputBuffer,
    [out]          PULONG ResultLength
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `CapabilityType`

Pointer to a GUID specifying the requested capability. The possible *PGUID* values are as follows:

- `GUID_USB_CAPABILITY_CHAINED_MDLS`
- `GUID_USB_CAPABILITY_STATIC_STREAMS`
- `GUID_USB_CAPABILITY_SELECTIVE_SUSPEND`
- `GUID_USB_CAPABILITY_FUNCTION_SUSPEND`
- `GUID_USB_CAPABILITY_DEVICE_CONNECTION_HIGH_SPEED_COMPATIBLE`
- `GUID_USB_CAPABILITY_DEVICE_CONNECTION_SUPER_SPEED_COMPATIBLE`

- `GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL`
  - For typical host controllers, the query must fail (`STATUS_NOT_SUPPORTED`). If the controller succeeds this capability, it is requesting a Clear TT Buffer when canceling Low Speed/Full Speed asynchronous (Bulk or Control) transfers that have been sent to a TT hub.

See the Remarks section of [USBD\\_QueryUsbCapability](#) for more information.

**[in] OutputBufferLength**

The length, in bytes, of the request's output buffer, if an output buffer is available.

**[out, optional] OutputBuffer**

A pointer to a location that receives the buffer's address. Certain capabilities may need to provide additional information to UCX in this buffer.

**[out] ResultLength**

A location that, on return, contains the size, in bytes, of the information that the callback function stored in *OutputBuffer*.

## Return value

If the operation is successful, the callback function must return `STATUS_SUCCESS`, or another status value for which `NT_SUCCESS(status)` equals TRUE. Otherwise it must return a status value for which `NT_SUCCESS(status)` equals FALSE.

Return code	Description
<code>STATUS_SUCCESS</code>	The requested USB capability is supported.
<code>STATUS_NOT_IMPLEMENTED</code>	The requested USB capability is unknown and not supported.
<code>STATUS_NOT_SUPPORTED</code>	Controller does not support the requested USB capability. For <code>GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL</code> , the controller did not request a Clear TT Buffer when canceling Low Speed/Full Speed asynchronous (Bulk or Control) transfers that were sent to a TT hub.

## Remarks

The UCX client driver registers its `EVT_UCX_CONTROLLER_QUERY_USB_CAPABILITY` implementation with the USB host controller extension (UCX) by calling the

[UcxControllerCreate](#) method.

## Examples

```
NTSTATUS
Controller_EvtControllerQueryUsbCapability(
    UCXCONTROLLER    UcxController,
    PGUID            CapabilityType,
    ULONG             OutputBufferLength,
    PVOID             OutputBuffer,
    PULONG            ResultLength
)
{
    NTSTATUS status;

    UNREFERENCED_PARAMETER(UcxController);
    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?
        //
        DbgTrace(TL_INFO, Controller, "GUID_USB_CAPABILITY_CHAINED_MDLS not
supported");
        status = STATUS_NOT_SUPPORTED;
    }
    else if (RtlCompareMemory(CapabilityType,
                               &GUID_USB_CAPABILITY_STATIC_STREAMS,
                               sizeof(GUID)) == sizeof(GUID)) {

        //
        // TODO: Is GUID_USB_CAPABILITY_STATIC_STREAMS supported?
        //
        DbgTrace(TL_INFO, Controller, "GUID_USB_CAPABILITY_STATIC_STREAMS
supported");
        status = STATUS_NOT_SUPPORTED;
    }
    else if (RtlCompareMemory(CapabilityType,
                               &GUID_USB_CAPABILITY_FUNCTION_SUSPEND,
                               sizeof(GUID)) == sizeof(GUID)) {

        //
        // TODO: Is GUID_USB_CAPABILITY_FUNCTION_SUSPEND supported?
        //
        DbgTrace(TL_INFO, Controller, "GUID_USB_CAPABILITY_FUNCTION_SUSPEND
```

```

        not supported");
            status = STATUS_NOT_SUPPORTED;
        }
        else if (RtlCompareMemory(CapabilityType,
                                    &GUID_USB_CAPABILITY_SELECTIVE_SUSPEND,
                                    sizeof(GUID)) == sizeof(GUID)) {

            DbgTrace(TL_INFO, Controller, "GUID_USB_CAPABILITY_SELECTIVE_SUSPEND
supported");
            status = STATUS_SUCCESS;
        }
        else if (RtlCompareMemory(CapabilityType,
&GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL,
                                    sizeof(GUID)) == sizeof(GUID)) {

            //
            // TODO: Is
GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL supported?
            //
            DbgTrace(TL_INFO, Controller,
"GUID_USB_CAPABILITY_CLEAR_TT_BUFFER_ON_ASYNC_TRANSFER_CANCEL not
supported");
            status = STATUS_NOT_SUPPORTED;
        }
        else {
            DbgTrace(TL_INFO, Controller, "Unhandled USB capability");
            status = STATUS_NOT_IMPLEMENTED;
        }

        return status;
    }
}

```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

# EVT\_UCX\_CONTROLLER\_RESET callback function (ucxcontroller.h)

Article02/22/2024

The client driver's implementation that UCX calls to reset the controller.

## Syntax

C++

```
EVT_UCX_CONTROLLER_RESET EvtUcxControllerReset;

void EvtUcxControllerReset(
    [in] UCXCONTROLLER UcxController
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

## Return value

None

## Remarks

The UCX client driver registers its *EVT\_UCX\_CONTROLLER\_RESET* implementation with the USB host controller extension (UCX) by calling the [UcxControllerCreate](#) method.

The client driver indicates completion of this event by calling the [UcxControllerResetComplete](#) method. Doing so ensures that UCX does not call *EVT\_UCX\_CONTROLLER\_RESET* a second time before this event callback completes.

If the client driver calls [UcxControllerNeedsReset](#), UCX calls this event callback function. However, UCX may call this event callback function even when the client driver has not

called **UcxControllerNeedsReset**.

## Examples

```
VOID
Controller_EvtControllerReset(
    UCXCONTROLLER UcxController
)

{
    UCX_CONTROLLER_RESET_COMPLETE_INFO controllerResetCompleteInfo;

    //
    // TODO: Reset the controller
    //

    //
    // TODO: Were devices and endpoints programmed in the controller before
    // the reset
    // still programmed in the controller after the reset?
    //
    UCX_CONTROLLER_RESET_COMPLETE_INFO_INIT(&controllerResetCompleteInfo,
                                            UcxControllerStateLost,
                                            TRUE); // reset due to UCX,
    received EvtReset after WDF power-up

    DbgTrace(TL_INFO, Controller, "Controller_EvtControllerReset");

    UcxControllerResetComplete(UcxController, &controllerResetCompleteInfo);
}
```

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

[UcxControllerNeedsReset](#)

[UcxControllerResetComplete](#)

# EVT\_UCX\_CONTROLLER\_SET\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_NOTIFICATION callback function (ucxcontroller.h)

Article02/22/2024

UCX invokes this callback function to specify its preference in transport characteristics for which the client driver must send notifications when changes occur.

## Syntax

C++

```
EVT_UCX_CONTROLLER_SET_TRANSPORT_CHARACTERISTICS_CHANGE_NOTIFICATION
EvtUcxControllerSetTransportCharacteristicsChangeNotification;

void EvtUcxControllerSetTransportCharacteristicsChangeNotification(
    [in] UCXCONTROLLER UcxController,
    [in] UCX_CONTROLLER_TRANSPORT_CHARACTERISTICS_CHANGE_FLAGS
ChangeNotificationFlags
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] ChangeNotificationFlags

A bitwise option of flags that indicate the type transport characteristics in which UCX is interested. The flags are defined in [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_FLAGS](#).

## Return value

None

## Remarks

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

The UCX client driver registers its implementation with the USB host controller extension (UCX) by calling the [UcxControllerCreate](#) method.

For efficient power consumption, UCX invokes this callback function to specify the transport characteristics in which UCX is interested.

It is likely that if the client driver keeps looking for changes in transport characteristics in the controller. This may result in a high power consumption and may be inefficient if there are no USB device drivers registered for that change notification. To optimize the power consumption, UCX invokes this callback function to let the client driver know if any clients are registered for changes. This callback function passes the change notification flags as parameter. If a flag is set, it indicates that there is at least one device driver registered. If the flag is not set, it means that there are no clients registered and hence the controller can optimize power.

### ⓘ Note

It is optional for controller drivers to register or make use of these callback functions. It is valid for a controller driver to invoke the change notification even if UCX has indicated that there are no clients registered for it.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

Requirement	Value
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

- [UcxControllerCreate](#)

# EVT\_UCX\_CONTROLLER\_START\_TRACKING\_FOR\_TIME\_SYNC callback function (ucxcontroller.h)

Article02/22/2024

UCX invokes this callback function to the start time tracking functionality in the controller.

## Syntax

C++

```
EVT_UCX_CONTROLLER_START_TRACKING_FOR_TIME_SYNC
EvtUcxControllerStartTrackingForTimeSync;

void EvtUcxControllerStartTrackingForTimeSync(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST WdfRequest,
    [in] size_t OutputBufferLength,
    [in] size_t InputBufferLength
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] WdfRequest

A framework request object that contains the request to start time tracking.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available. This value is the size of the [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available. This value is the size of the [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## Return value

None

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

# EVT\_UCX\_CONTROLLER\_STOP\_TRACKING\_FOR\_TIME\_SYNC callback function (ucxcontroller.h)

Article 02/22/2024

UCX invokes this callback function to the stop time tracking functionality in the controller.

## Syntax

C++

```
EVT_UCX_CONTROLLER_STOP_TRACKING_FOR_TIME_SYNC
EvtUcxControllerStopTrackingForTimeSync;

void EvtUcxControllerStopTrackingForTimeSync(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST WdfRequest,
    [in] size_t OutputBufferLength,
    [in] size_t InputBufferLength
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] WdfRequest

A framework request object that contains the request to stop time tracking.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available. This value is the size of the [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available. This value is the size of the [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## Return value

None

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

# EVT\_UCX\_CONTROLLER\_USBDEVICE\_ADD callback function (ucxcontroller.h)

Article 02/22/2024

The client driver's implementation that UCX calls when a new USB device is detected.

## Syntax

C++

```
EVT_UCX_CONTROLLER_USBDEVICE_ADD EvtUcxControllerUsbdeviceAdd;

NTSTATUS EvtUcxControllerUsbdeviceAdd(
    [in] UCXCONTROLLER UcxController,
    [in] PUCXUSBDEVICE_INFO UcxUsbDeviceInfo,
    [in] PUCXUSBDEVICE_INIT UsbDeviceInit
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `UcxUsbDeviceInfo`

Pointer to a [UCXUSBDEVICE\\_INFO](#) structure.

[in] `UsbDeviceInit`

Pointer to an opaque structure containing initialization information. Callbacks for the device object are associated with this structure. This structure is managed by UCX.

## Return value

If the operation is successful, the callback function must return `STATUS_SUCCESS`, or another status value for which `NT_SUCCESS(status)` equals `TRUE`. Otherwise it must return a status value for which `NT_SUCCESS(status)` equals `FALSE`.

# Remarks

The UCX client driver registers its *EVT\_UCX\_CONTROLLER\_USBDEVICE\_ADD* implementation with the USB host controller extension (UCX) by calling the [UcxControllerCreate](#) method.

This callback function creates a new USB device object and registers the USB device object callback functions by calling [UcxUsbDeviceCreate](#). The function may need to allocate the common buffer that will be used as the device context.

## Examples

```
NTSTATUS
UsbDevice_EvtControllerUsbDeviceAdd(
    UCXCONTROLLER        UcxController,
    PUCXUSBDEVICE_INFO   UsbDeviceInfo,
    PUCXUSBDEVICE_INIT   UsbDeviceInit
)

{
    NTSTATUS                  status = STATUS_SUCCESS;

    WDF_OBJECT_ATTRIBUTES      objectAttributes;
    UCX_USBDEVICE_EVENT_CALLBACKS callbacks;

    UCXUSBDEVICE              ucxUsbDevice;
    PUCX_USB_DEVICE_CONTEXT   ucxUsbDeviceContext;

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes,
    UCX_USB_DEVICE_CONTEXT);

    //
    // Set the event callbacks for the USB device.
    //
    UCX_USBDEVICE_EVENT_CALLBACKS_INIT(&callbacks,

    UsbDevice_EvtUcxUsbDeviceEndpointsConfigure,
                                UsbDevice_EvtUcxUsbDeviceEnable,
                                UsbDevice_EvtUcxUsbDeviceDisable,
                                UsbDevice_EvtUcxUsbDeviceReset,
                                UsbDevice_EvtUcxUsbDeviceAddress,
                                UsbDevice_EvtUcxUsbDeviceUpdate,
                                UsbDevice_EvtUcxUsbDeviceHubInfo,

    Endpoint_EvtUcxUsbDeviceDefaultEndpointAdd,
                                Endpoint_EvtUcxUsbDeviceEndpointAdd);

    UcxUsbDeviceInitSetEventCallbacks(UsbDeviceInit, &callbacks);
```

```

// Create the device
//
status = UcxUsbDeviceCreate(UcxController,
                            &UsbDeviceInit,
                            &objectAttributes,
                            &ucxUsbDevice);

if (!NT_SUCCESS(status)) {
    DbgTrace(TL_ERROR, UsbDevice, "UcxUsbDeviceCreate Failed %!STATUS!", status);
    goto EvtControllerUsbDeviceAddEnd;
}

ucxUsbDeviceContext = GetUcxUsbDeviceContext(ucxUsbDevice);
ucxUsbDeviceContext->DeviceSpeed = UsbDeviceInfo->DeviceSpeed;
ucxUsbDeviceContext->TtHub = UsbDeviceInfo->TtHub;
RtlCopyMemory(&ucxUsbDeviceContext->PortPath,
              &UsbDeviceInfo->PortPath,
              sizeof(USB_DEVICE_PORT_PATH));

DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtControllerUsbDeviceAdd");

EvtControllerUsbDeviceAddEnd:

return status;
}

```

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

# UCX\_CONTROLLER\_ACPI\_INFORMATION structure (ucxcontroller.h)

Article02/22/2024

This structure provides information about an advanced Configuration and power interface (ACPI) USB controller.

## Syntax

C++

```
typedef struct _UCX_CONTROLLER_ACPI_INFORMATION {
    CHAR VendorId[MAX_VENDOR_ID_STRING_LENGTH];
    CHAR DeviceId[MAX_DEVICE_ID_STRING_LENGTH];
    CHAR RevisionId[MAX_REVISION_ID_STRING_LENGTH];
} UCX_CONTROLLER_ACPI_INFORMATION, *PUCX_CONTROLLER_ACPI_INFORMATION;
```

## Members

`VendorId[MAX_VENDOR_ID_STRING_LENGTH]`

The vendor ID of the ACPI USB controller.

`DeviceId[MAX_DEVICE_ID_STRING_LENGTH]`

The device ID of the ACPI USB controller.

`RevisionId[MAX_REVISION_ID_STRING_LENGTH]`

The revision ID of the ACPI USB controller.

## Requirements

[+] Expand table

Requirement	Value
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[UCX\\_CONTROLLER\\_CONFIG](#)

[UCX\\_CONTROLLER\\_CONFIG\\_SET\\_ACPI\\_INFO](#)

# UCX\_CONTROLLER\_CONFIG structure (ucxcontroller.h)

Article 02/22/2024

This structure configuration data for a USB controller.

## Syntax

C++

```
typedef struct _UCX_CONTROLLER_CONFIG {
    ULONG                                         Size;
    ULONG                                         NumberofPresentedDeviceMgmtEvtCallbacks;
    PFN_UCX_CONTROLLER_QUERY_USB_CAPABILITY      EvtControllerQueryUsbCapability;
    HANDLE                                         Reserved1;
    PFN_UCX_CONTROLLER_GET_CURRENT_FRAMENUMBER   EvtControllerGetCurrentFrameNumber;
    PFN_UCX_CONTROLLER_USBDEVICE_ADD              EvtControllerUsbDeviceAdd;
    PFN_UCX_CONTROLLER_RESET                      EvtControllerReset;
    HANDLE                                         Reserved2;
    HANDLE                                         Reserved3;
    HANDLE                                         Reserved4;
    UCX_CONTROLLER_PARENT_BUS_TYPE                ParentBusType;
    UCX_CONTROLLER_PCI_INFORMATION               PciDeviceInfo;
    UCX_CONTROLLER_ACPI_INFORMATION              AcpiDeviceInfo;
    UCHAR                                         DeviceDescription[MAX_GENERIC_USB_CONTROLLER_NAME_SIZE];
    UNICODE_STRING                                ManufacturerNameString;
    UNICODE_STRING                                ModelNameString;
    UNICODE_STRING                                ModelNumberString;
    PFN_UCX_CONTROLLER_GET_TRANSPORT_CHARACTERISTICS EvtControllerGetTransportCharacteristics;
    PFN_UCX_CONTROLLER_SET_TRANSPORT_CHARACTERISTICS_CHANGE_NOTIFICATION
    EvtControllerSetTransportCharacteristicsChangeNotification;
    HANDLE                                         
```

```
Reserved5;
    HANDLE
Reserved6;
    HANDLE
Reserved7;
} UCX_CONTROLLER_CONFIG, *PUCX_CONTROLLER_CONFIG;
```

## Members

Size

The size in bytes of this structure.

NumberOfPresentedDeviceMgmtEvtCallbacks

The number of device event callback functions provided by this structure.

EvtControllerQueryUsbCapability

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_QUERY\\_USB\\_CAPABILITY](#) callback function.

Reserved1

Do not use.

EvtControllerGetCurrentFrameNumber

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_GET\\_CURRENT\\_FRAMENUMBER](#) call back function.

EvtControllerUsbDeviceAdd

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#) callback function.

EvtControllerReset

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_RESET](#) callback function.

Reserved2

Do not use.

Reserved3

Do not use.

Reserved4

Do not use.

`ParentBusType`

The parent bus type of the USB controller.

`PciDeviceInfo`

Information about the PCI USB controller (if present).

`AcpiDeviceInfo`

Information about the advanced configuration and power interface (ACPI) USB controller (if present).

`DeviceDescription[MAX_GENERIC_USB_CONTROLLER_NAME_SIZE]`

A description for the device.

`ManufacturerNameString`

String containing the manufacturer name.

`ModelNameString`

String containing the model name of the controller hardware.

`ModelNumberString`

String containing the model number of the controller hardware.

`EvtControllerGetTransportCharacteristics`

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#) callback function.

`EvtControllerSetTransportCharacteristicsChangeNotification`

A pointer to an [EVT\\_UCX\\_CONTROLLER\\_SET\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#) callback function.

`Reserved5`

Do not use.

`Reserved6`

Do not use.

Reserved7

Do not use.

# Requirements

 Expand table

Requirement	Value
Header	ucxcontroller.h (include Ucxclass.h)

# UCX\_CONTROLLER\_CONFIG\_SET\_ACPI\_INFO function (ucxcontroller.h)

Article 02/22/2024

Initializes a [UCX\\_CONTROLLER\\_CONFIG](#) structure with the specified values for the controller with ACPI as the parent.

## Syntax

C++

```
void UCX_CONTROLLER_CONFIG_SET_ACPI_INFO(
    PUCX_CONTROLLER_CONFIG Config,
    PSTR                 VendorId,
    PSTR                 DeviceId,
    PSTR                 RevisionId
);
```

## Parameters

`Config`

A pointer to a [UCX\\_CONTROLLER\\_CONFIG](#) structure to initialize.

`VendorId`

A string that contains the vendor identifier for the device.

`DeviceId`

A string that specifies the device identifier assigned by the manufacturer.

`RevisionId`

A string that specifies the revision level of the device described by the `DeviceID` member.

## Return value

None

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[UCX\\_CONTROLLER\\_CONFIG](#)

[UcxControllerCreate](#)

# UCX\_CONTROLLER\_CONFIG\_SET\_PCI\_IN FO function (ucxcontroller.h)

Article02/22/2024

Initializes a [UCX\\_CONTROLLER\\_CONFIG](#) structure with the specified values for the controller with PCI as the parent bus type.

## Syntax

C++

```
void UCX_CONTROLLER_CONFIG_SET_PCI_INFO(
    [in] PUCX_CONTROLLER_CONFIG Config,
    [in] ULONG             VendorId,
    [in] ULONG             DeviceId,
    [in] USHORT            RevisionId,
    [in] ULONG             BusNumber,
    [in] ULONG             DeviceNumber,
    [in] ULONG             FunctionNumber
);
```

## Parameters

[in] Config

A pointer to a [UCX\\_CONTROLLER\\_CONFIG](#) structure to initialize.

[in] VendorId

Specifies the vendor identifier for the device as assigned by the PCI SIG.

[in] DeviceId

Specifies the device identifier assigned by the manufacturer.

[in] RevisionId

Specifies the revision level of the device described by the DeviceID member.

[in] BusNumber

Specifies the bus number that identifies the bus instance that a device instance is attached to.

[in] DeviceNumber

Specifies the device number that is assigned to the logical PCI slot.

[in] FunctionNumber

Specifies the specific function on the device that is located in the logical PCI slot.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[UCX\\_CONTROLLER\\_CONFIG](#)

[UcxControllerCreate](#)

# UCX\_CONTROLLER\_PARENT\_BUS\_TYPE enumeration (ucxcontroller.h)

Article 02/22/2024

The UCX\_CONTROLLER\_PARENT\_BUS\_TYPE enumeration defines the parent bus type.

## Syntax

C++

```
typedef enum _UCX_CONTROLLER_PARENT_BUS_TYPE {  
    UcxControllerParentBusTypeCustom,  
    UcxControllerParentBusTypePci,  
    UcxControllerParentBusTypeAcpi,  
    UcxControllerParentBusTypeMaUsb  
} UCX_CONTROLLER_PARENT_BUS_TYPE;
```

## Constants

[ ] Expand table

<code>UcxControllerParentBusTypeCustom</code>	Custom bus type.
<code>UcxControllerParentBusTypePci</code>	Parent bus is PCI.
<code>UcxControllerParentBusTypeAcpi</code>	Parent is ACPI.
<code>UcxControllerParentBusTypeMaUsb</code>	

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[UCX\\_CONTROLLER\\_CONFIG](#)

# UCX\_CONTROLLER\_PCI\_INFORMATION structure (ucxcontroller.h)

Article04/01/2021

This structure provides information about a PCI USB controller.

## Syntax

C++

```
typedef struct _UCX_CONTROLLER_PCI_INFORMATION {
    ULONG VendorId;
    ULONG DeviceId;
    USHORT RevisionId;
    ULONG BusNumber;
    ULONG DeviceNumber;
    ULONG FunctionNumber;
} UCX_CONTROLLER_PCI_INFORMATION, *PUCX_CONTROLLER_PCI_INFORMATION;
```

## Members

**VendorId**

The vendor ID for the PCI USB controller.

**DeviceId**

The device ID for the PCI USB controller.

**RevisionId**

The revision ID for the PCI USB controller.

**BusNumber**

Specifies the bus number that identifies the bus instance that a device instance is attached to.

**DeviceNumber**

Specifies the device number that is assigned to the logical PCI slot.

**FunctionNumber**

Specifies the specific function on the device that is located in the logical PCI slot.

## Requirements

Header	ucxcontroller.h (include Ucxclass.h)
--------	--------------------------------------

## See also

[UCX\\_CONTROLLER\\_CONFIG](#)

[UCX\\_CONTROLLER\\_CONFIG\\_SET\\_PCI\\_INFO](#)

[UCX\\_CONTROLLER\\_PARENT\\_BUS\\_TYPE](#)

# UCX\_CONTROLLER\_RESET\_COMPLETE\_INFO structure (ucxcontroller.h)

Article02/22/2024

Contains information about the operation to reset the controller. This is used by the client driver in its [EVT\\_UCX\\_CONTROLLER\\_RESET](#) callback function.

## Syntax

C++

```
typedef struct _UCX_CONTROLLER_RESET_COMPLETE_INFO {
    ULONG             Size;
    UCX_CONTROLLER_STATE UcxControllerState;
    BOOLEAN           UcxCoordinated;
} UCX_CONTROLLER_RESET_COMPLETE_INFO, *PUCX_CONTROLLER_RESET_COMPLETE_INFO;
```

## Members

Size

The size in bytes of this structure.

UcxControllerState

The UCX controller state after reset.

UcxCoordinated

Indicates if the reset was coordinated with UCX (TRUE) or not (FALSE).

## Remarks

This structure is populated by a call to [UcxControllerResetComplete](#).

## Requirements

[ ] [Expand table](#)

<b>Requirement</b>	<b>Value</b>
Header	ucxcontroller.h (include Ucxclass.h)

# UCX\_CONTROLLER\_STATE enumeration (ucxcontroller.h)

Article 02/22/2024

This enumeration provides values to specify the UCX controller state after a reset.

## Syntax

C++

```
typedef enum _UCX_CONTROLLER_STATE {  
    UcxControllerStateLost,  
    UcxControllerStatePreserved  
} UCX_CONTROLLER_STATE;
```

## Constants

  Expand table

<b>UcxControllerStateLost</b>
Indicates the controller state was lost after reset.
<b>UcxControllerStatePreserved</b>
Indicates the controller state was preserved after reset.

## Requirements

  Expand table

Requirement	Value
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[UCX\\_CONTROLLER\\_RESET\\_COMPLETE\\_INFO](#)

# UCX\_CONTROLLER\_TRANSPORT\_CHARACTERISTICS structure (ucxcontroller.h)

Article 02/22/2024

Stores the transport characteristics at relevant points in time. This structure is used in the [EVT\\_UCX\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#) callback function.

## Syntax

C++

```
typedef struct _UCX_CONTROLLER_TRANSPORT_CHARACTERISTICS {
    ULONG TransportCharacteristicsFlags;
    ULONG64 CurrentRoundtripLatencyInMilliSeconds;
    ULONG64 MaxPotentialBandwidth;
} UCX_CONTROLLER_TRANSPORT_CHARACTERISTICS,
*PUCX_CONTROLLER_TRANSPORT_CHARACTERISTICS;
```

## Members

TransportCharacteristicsFlags

A bitmask that indicates to the client driver the transport characteristics that are available and are returned in this structure.

If **USB\_TRANSPORT\_CHARACTERISTICS\_LATENCY\_AVAILABLE**

is set, **CurrentRoundtripLatencyInMilliSeconds** contains valid information. Otherwise , it must not be used by the client driver.

If **USB\_TRANSPORT\_CHARACTERISTICS\_BANDWIDTH\_AVAILABLE**

is set, **MaxPotentialBandwidth** contains valid information. Otherwise, it must not be used by the client driver.

CurrentRoundtripLatencyInMilliSeconds

Contains the current round-trip delay in milliseconds from the time a non-isochronous transfer is received by the USB driver stack to the time that the transfer is completed.

For MA-USB, the underlying network could be WiFi, WiGig, Ethernet etc. The delay can vary depending on the underlying network conditions. A client driver should query the

latency periodically or whenever it is notified of a change.

#### MaxPotentialBandwidth

Contains the total bandwidth of the host controller's shared transport.

For MA-USB, the underlying network transport could be WiFi, WiGig, Ethernet etc. The total available bandwidth can vary depending on several factors such as the negotiation WiFi channel. A client driver should query the total bandwidth periodically or whenever it is notified of a change.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

# UCX\_CONTROLLER\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_FLAGS union (ucxcontroller.h)

Article 02/22/2024

Defines flags for the transport characteristics changes. This structure is used in the [EVT\\_UCX\\_CONTROLLER\\_SET\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#) callback function.

## Syntax

C++

```
typedef union _UCX_CONTROLLER_TRANSPORT_CHARACTERISTICS_CHANGE_FLAGS {
    ULONG AsUlong32;
    struct {
        ULONG CurrentRoundtripLatencyChanged : 1;
        ULONG CurrentTotalBandwidthChanged : 1;
    } Flags;
    struct {
        ULONG CurrentRoundtripLatencyChanged : 1;
        ULONG CurrentTotalBandwidthChanged : 1;
    };
} UCX_CONTROLLER_TRANSPORT_CHARACTERISTICS_CHANGE_FLAGS;
```

## Members

AsUlong32

Reserved.

Flags

Flags.CurrentRoundtripLatencyChanged

Flags.CurrentTotalBandwidthChanged

CurrentRoundtripLatencyChanged

Contains the current round-trip delay in milliseconds from the time a non-isochronous transfer is received by the USB driver stack to the time that the transfer is completed.

For MA-USB, the underlying network could be WiFi, WiGig, Ethernet etc. The delay can vary depending on the underlying network conditions. A client driver should query the latency periodically or whenever it is notified of a change.

#### CurrentTotalBandwidthChanged

Contains the total bandwidth of the host controller's shared transport.

For MA-USB, the underlying network transport could be WiFi, WiGig, Ethernet etc. The total available bandwidth can vary depending on several factors such as the negotiation WiFi channel. A client driver should query the total bandwidth periodically or whenever it is notified of a change.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucxcontroller.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_CONTROLLER\\_SET\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#)

# UcxControllerCreate function (ucxcontroller.h)

Article 02/22/2024

Creates a host controller object.

## Syntax

C++

```
NTSTATUS UcxControllerCreate(
    [in]          WDFDEVICE             Device,
    [in]          PUCX_CONTROLLER_CONFIG Config,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]         UCXCONTROLLER        *Controller
);
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] Config

A pointer to a caller-allocated [UCX\\_CONTROLLER\\_CONFIG](#) structure that the client driver initialized by calling [UCX\\_CONTROLLER\\_CONFIG\\_INIT](#). The structure contains configuration information required to create the object.

[in, optional] Attributes

A pointer to a caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that specifies attributes for the controller object.

[out] Controller

A pointer to a variable that receives a handle to the new controller object.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return one an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the host controller must call this method after the [WdfDeviceCreate](#) call. The parent of the new host controller object is the framework device object.

During this call, the client driver-supplied event callback implementations are also registered. Supply function pointers to those functions by call setting appropriate members of [UCX\\_CONTROLLER\\_CONFIG](#).

If the parent type is PCI, then initialize the [UCX\\_CONTROLLER\\_CONFIG](#) by calling [UCX\\_CONTROLLER\\_CONFIG\\_SET\\_PCI\\_INFO](#). If the parent is ACPI, call [UCX\\_CONTROLLER\\_CONFIG\\_SET\\_ACPI\\_INFO](#) instead.

The method creates various queues required to handle IOCTL requests sent to the USB device.

The method registers a device interface `GUID_DEVINTERFACE_USB_HOST_CONTROLLER` and symbolic link so that user mode components can open a handle to the controller.

## Examples

```
WDF_OBJECT_ATTRIBUTES          objectAttributes;
UCX_CONTROLLER_CONFIG          ucxControllerConfig;
UCXCONTROLLER                 ucxController;
PUCX_CONTROLLER_CONTEXT        ucxControllerContext;

// Create the controller
//
UCX_CONTROLLER_CONFIG_INIT(&ucxControllerConfig, "");
ucxControllerConfig.EvtControllerUsbDeviceAdd =
UsbDevice_EvtControllerUsbDeviceAdd;
ucxControllerConfig.EvtControllerQueryUsbCapability =
Controller_EvtControllerQueryUsbCapability;
ucxControllerConfig.EvtControllerReset = Controller_EvtControllerReset;
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes,
UCX_CONTROLLER_CONTEXT);

status = UcxControllerCreate(wdfDevice,
    &ucxControllerConfig,
    &objectAttributes,
    &ucxController);
```

```

if (!NT_SUCCESS(status)) {
    DbgTrace(TL_ERROR, Controller, "UcxControllerCreate Failed
%!STATUS!", status);
    goto Controller_WdfEvtDeviceAddEnd;
}

DbgTrace(TL_INFO, Controller, "UCX Controller created.");

controllerContext->UcxController = ucxController;

ucxControllerContext = GetUcxControllerContext(ucxController);
ucxControllerContext->WdfDevice = wdfDevice;

```

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[WdfDeviceCreate](#)

# UcxControllerNeedsReset function (ucxcontroller.h)

Article02/22/2024

Initiates a non-Plug and Play (PnP) controller reset operation by queuing an event into the controller reset state machine.

## Syntax

C++

```
void UcxControllerNeedsReset(
    [in] UCXCONTROLLER Controller
);
```

## Parameters

[in] Controller

A handle to the controller object to reset. The client driver retrieved the handle in a previous call to [UcxControllerCreate](#).

## Return value

None

## Remarks

If the operation is successful, the method returns TRUE. Otherwise, it returns FALSE.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

- [UcxControllerCreate](#)

# UcxControllerNotifyTransportCharacteristicsChange function (ucxcontroller.h)

Article 02/22/2024

Notifies UCX about a new port change event from host controller.

## Syntax

C++

```
void UcxControllerNotifyTransportCharacteristicsChange(
    UCXCONTROLLER Controller,
    [out] PUCX_CONTROLLER_TRANSPORT_CHARACTERISTICS
    UcxControllerTransportCharacteristics
);
```

## Parameters

Controller

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[out] UcxControllerTransportCharacteristics

A pointer to a [UCX\\_CONTROLLER\\_TRANSPORT\\_CHARACTERISTICS](#) structure that contains the updated transport characteristics.

## Return value

None

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
Library	Ucxstubs.lib

## See also

[EVT\\_UCX\\_CONTROLLER\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

# UcxControllerResetComplete function (ucxcontroller.h)

Article02/22/2024

Informs USB host controller extension (UCX) that the reset operation has completed.

## Syntax

C++

```
void UcxControllerResetComplete(
    [in] UCXCONTROLLER Controller,
    PUCX_CONTROLLER_RESET_COMPLETE_INFO UcxControllerResetCompleteInfo
);
```

## Parameters

[in] Controller

A handle to the controller object to reset. The client driver retrieved the handle in a previous call to [UcxControllerCreate](#).

UcxControllerResetCompleteInfo

Pointer to information about the UCX controller state after the reset completes.

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows

Requirement	Value
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxControllerSetFailed function (ucxcontroller.h)

Article02/22/2024

Informs USB Host Controller Extension (UCX) that the controller has encountered a critical failure.

## Syntax

C++

```
void UcxControllerSetFailed(  
    [in] UCXCONTROLLER Controller  
);
```

## Parameters

[in] Controller

A handle to the controller object. The client driver retrieved the handle in a previous call to [UcxControllerCreate](#).

## Return value

None

## Remarks

The client driver for host controller must call this function if the driver fails D0 entry or the driver has stopped processing transfers to or from an endpoint.

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Target Platform	Windows
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxControllerCreate](#)

# UcxControllerSetIdStrings function (ucxcontroller.h)

Article 02/22/2024

Updates the identifier strings of a controller after the controller has been initialized.

## Syntax

C++

```
NTSTATUS UcxControllerSetIdStrings(
    UCXCONTROLLER Controller,
    [in] PUNICODE_STRING ManufacturerNameString,
    [in] PUNICODE_STRING ModelNameString,
    [in] PUNICODE_STRING ModelNumberString
);
```

## Parameters

`Controller`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

`[in] ManufacturerNameString`

A string that contains the name of controller manufacturer.

`[in] ModelNameString`

A string that contains the name of device model.

`[in] ModelNumberString`

A string that contains the revision number of the device model.

## Return value

The function returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return one an appropriate [NTSTATUS](#) error code.

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxcontroller.h (include Ucxclass.h)
Library	Ucxstubs.lib

# UcxIoDeviceControl function (ucxcontroller.h)

Article 10/21/2021

Allows USB host controller extension (UCX) to handle an I/O control code (IOCTL) request from user mode.

## Syntax

C++

```
BOOLEAN UcxIoDeviceControl(
    [in] WDFDEVICE Device,
    [in] WDFREQUEST Request,
    [in] size_t OutputBufferLength,
    [in] size_t InputBufferLength,
    [in] ULONG IoControlCode
);
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] Request

A handle to a framework request object that represents the user-mode IOCTL request.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available.

[in] IoControlCode

The driver-defined or system-defined IOCTL that is associated with the request.

# Return value

If the operation is successful, the method returns TRUE. Otherwise it returns FALSE.

## Remarks

The client driver can call this method to allow UCX to handle IOCTLs listed in this table: [User-Mode IOCTLs for USB](#). If the IOCTL code is [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_OFF](#) or [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_ON](#), UCX completes the request successfully. For IOCTLs that are used to retrieve the USB host controllers driver key name, such as [IOCTL\\_USB\\_GET\\_ROOT\\_HUB\\_NAME](#) or [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#), UCX retrieves the Unicode string. If the user mode IOCTL is [IOCTL\\_USB\\_USER\\_REQUEST](#), the input and output buffer lengths must be equal and the output buffer must contain the [USBUSER\\_REQUEST\\_HEADER](#) structure. For the remaining IOCTLs, UCX returns FALSE and the client driver can provide its own handling logic.

## Examples

```
VOID
Controller_WdfEvtIoDeviceControl(
    WDFQUEUE     WdfQueue,
    WDFREQUEST   WdfRequest,
    size_t        OutputBufferLength,
    size_t        InputBufferLength,
    ULONG         IoControlCode
)
/*++
```

### Routine Description:

This routine is a callback function which is called by WDF when a driver receives an I/O control request from the queue this callback is registered with.

The controller driver calls `UcxIoDeviceControl()` to allow UCX to try and handle the IOCTL. If UCX cannot handle the IOCTL, the controller driver must handle it, perhaps by failing it.

The default queue only expects to receive IOCTLs from user mode (via the interface defined by `GUID_DEVINTERFACE_USB_HOST_CONTROLLER`).

### Arguments:

`WdfQueue` - A handle to the framework I/O queue object.

WdfRequest - A handle to the framework request object that contains the IOCTL.

OutputBufferLength - Length of the IOCTL output buffer, if an output buffer is available.

InputBufferLength - Length of the IOCTL input buffer, if an input buffer is available.

IoControlCode - I/O control code associated with the request.

Return Value:

None.

```
--*/
{
    KPROCESSOR_MODE requestorMode;

    //
    // Allow UCX to try and handle the request
    //
    if (UcxIoDeviceControl(WdfIoQueueGetDevice(WdfQueue),
                           WdfRequest,
                           OutputBufferLength,
                           InputBufferLength,
                           IoControlCode)) {
        DbgTrace(TL_VERBOSE, Controller, "IoControlCode 0x%x was handled by
UCX", IoControlCode);
        goto WdfEvtIoDeviceControlEnd;
    }

    //
    // Check that the request is coming from user mode
    //
    requestorMode = WdfRequestGetRequestorMode(WdfRequest);

    if (requestorMode != UserMode) {
        DbgTrace(TL_WARNING, Controller, "Invalid RequestorMode %d",
requestorMode);
    }

    //
    // UCX could not handle the request, so handle it here
    //
    switch (IoControlCode) {

        default:
            DbgTrace(TL_WARNING, Controller, "Unsupported IoControlCode 0x%x",
IoControlCode);
            WdfRequestComplete(WdfRequest, STATUS_INVALID_DEVICE_REQUEST);
    }
}
```

```
WdfEvtIoDeviceControlEnd:
```

```
    return;  
}
```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ucxcontroller.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

[User-Mode IOCTLs for USB](#)

# ucxendpoint.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucxendpoint.h contains the following programming interfaces:

## Functions

### [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

Initializes a UCX\_DEFAULT\_ENDPOINT\_EVENT\_CALLBACKS structure with client driver's callback functions. The client driver calls this function before calling UcxEndpointCreate method to create an endpoint and register its callback functions with UCX.

### [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

Initializes a UCX\_ENDPOINT\_EVENT\_CALLBACKS structure with client driver's callback functions. The client driver calls this function before calling UcxEndpointCreate method to create an endpoint and register its callback functions with UCX.

### [UcxDefaultEndpointInitSetEventCallbacks](#)

Initializes a UCXENDPOINT\_INIT structure with client driver's event callback functions related to the default endpoint.

### [UcxEndpointAbortComplete](#)

Notifies UCX that a transfer abort operation has been completed on the specified endpoint object.

### [UcxEndpointCreate](#)

Creates an endpoint on the specified USB device object.

### [UcxEndpointGetStaticStreamsReferenced](#)

Returns a referenced static streams object for the specified endpoint.

### [UcxEndpointInitSetEventCallbacks](#)

Initializes a UCXENDPOINT\_INIT structure with client driver's event callback functions related to endpoints on the device.

## [UcxEndpointNeedToCancelTransfers](#)

The client driver calls this method before it cancels transfers on the wire.

## [UcxEndpointNoPingResponseError](#)

Notifies UCX about a "No Ping Response" error for a transfer on the specified endpoint object.

## [UcxEndpointPurgeComplete](#)

Notifies UCX that a purge operation has been completed on the specified endpoint object.

## [UcxEndpointSetWdfIoQueue](#)

Sets a framework queue on the specified endpoint object.

# Callback functions

## [EVT\\_UCX\\_DEFAULT\\_ENDPOINT\\_UPDATE](#)

The client driver's implementation that UCX calls with information about the default endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_ABORT](#)

The client driver's implementation that UCX calls to abort the queue associated with the endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_GET\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#)

UCX invokes this callback function to get information about transfer path delays for an isochronous endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_OK\\_TO\\_CANCEL\\_TRANSFERS](#)

The client driver's implementation that UCX calls to notify the controller driver that it can complete cancelled transfers on the endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_PURGE](#)

The client driver's implementation that completes all outstanding I/O requests on the endpoint.

## [EVT\\_UCX\\_ENDPOINT\\_RESET](#)

The client driver's implementation that UCX calls to reset the controller's programming for an endpoint.

## EVT\_UCX\_ENDPOINT\_SET\_CHARACTERISTIC

UCX invokes this callback function to set the priority on an endpoint.

## EVT\_UCX\_ENDPOINT\_START

The client driver's implementation that UCX calls to start the queue associated with the endpoint.

## EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_ADD

The client driver's implementation that UCX calls to create static streams.

## EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_DISABLE

The client driver's implementation that UCX calls to release controller resources for all streams for an endpoint.

## EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_ENABLE

The client driver's implementation that UCX calls to enable the static streams.

# Structures

## DEFAULT\_ENDPOINT\_UPDATE

Contains the handle to the default endpoint to update in a framework request that is passed by UCX when it invokes EVT\_UCX\_DEFAULT\_ENDPOINT\_UPDATE callback function.

## ENDPOINT\_RESET

Describes information required to reset an endpoint. This structure is passed by UCX in the EVT\_UCX\_ENDPOINT\_RESET callback function.

## ENDPOINTS\_CONFIGURE

Describes endpoints to enable or disable endpoints. This structure is passed by UCX in the EVT\_UCX\_USBDEVICE\_ENDPOINTS\_CONFIGURE callback function.

## ENDPOINTS\_CONFIGURE\_FAILURE\_FLAGS

This structure provides failure flags to indicate errors, if any, that might have occurred during a request to an EVT\_UCX\_USBDEVICE\_ENDPOINTS\_CONFIGURE callback function.

## UCX\_DEFAULT\_ENDPOINT\_EVENT\_CALLBACKS

This structure provides a list of UCX default endpoint event callback functions.

#### [UCX\\_ENDPOINT\\_CHARACTERISTIC](#)

Stores the characteristics of an endpoint.

#### [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS](#)

This structure provides a list of pointers to UCX endpoint event callback functions.

#### [UCX\\_ENDPOINT\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#)

Stores the isochronous transfer path delay values.

## Enumerations

#### [ENDPOINT\\_RESET\\_FLAGS](#)

Defines parameters for a request to reset an endpoint.

#### [UCX\\_CONTROLLER\\_ENDPOINT\\_CHARACTERISTIC\\_PRIORITY](#)

Indicates the priority of endpoints.

#### [UCX\\_ENDPOINT\\_CHARACTERISTIC\\_TYPE](#)

Defines values that indicates the type of endpoint characteristic.

# DEFAULT\_ENDPOINT\_UPDATE structure (ucxendpoint.h)

Article02/22/2024

Contains the handle to the default endpoint to update in a framework request that is passed by UCX when it invokes [EVT\\_UCX\\_DEFAULT\\_ENDPOINT\\_UPDATE](#) callback function.

## Syntax

C++

```
typedef struct _DEFAULT_ENDPOINT_UPDATE {
    USBDEVICE_MGMT_HEADER Header;
    UCXENDPOINT          DefaultEndpoint;
    ULONG                MaxPacketSize;
} DEFAULT_ENDPOINT_UPDATE, *PDEFAULT_ENDPOINT_UPDATE;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains header information related to the USB device or hub endpoint.

DefaultEndpoint

A handle to the default endpoint to update.

MaxPacketSize

The maximum packet size of the default endpoint.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[EVT\\_UCX\\_DEFAULT\\_ENDPOINT\\_UPDATE](#)

# ENDPOINT\_RESET structure (ucxendpoint.h)

Article 02/22/2024

Describes information required to reset an endpoint. This structure is passed by UCX in the [EVT\\_UCX\\_ENDPOINT\\_RESET](#) callback function.

## Syntax

C++

```
typedef struct _ENDPOINT_RESET {
    USBDEVICE_MGMT_HEADER Header;
    UCXENDPOINT          Endpoint;
    ENDPOINT_RESET_FLAGS Flags;
} ENDPOINT_RESET, *PENDPOINT_RESET;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that stores handles to the USB hub or device whose endpoints.

Endpoint

A handle to the device endpoint to reset.

Flags

A [ENDPOINT\\_RESET\\_FLAGS](#) value that indicates reset parameters.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[ENDPOINT\\_RESET\\_FLAGS](#)

# ENDPOINT\_RESET\_FLAGS enumeration (ucxendpoint.h)

Article02/22/2024

Defines parameters for a request to reset an endpoint.

## Syntax

C++

```
typedef enum _ENDPOINT_RESET_FLAGS {
    FlagEndpointResetPreserveTransferState
} ENDPOINT_RESET_FLAGS;
```

## Constants

[ ] Expand table

FlagEndpointResetPreserveTransferState

The transfer state must be preserved after the endpoint reset operation is complete.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[ENDPOINT\\_RESET](#)

[EVT\\_UCX\\_ENDPOINT\\_RESET](#)

# ENDPOINTS\_CONFIGURE structure (ucxendpoint.h)

Article 02/22/2024

Describes endpoints to enable or disable endpoints. This structure is passed by UCX in the [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback function.

## Syntax

C++

```
typedef struct _ENDPOINTS_CONFIGURE {
    USBDEVICE_MGMT_HEADER           Header;
    ULONG                           EndpointsToEnableCount;
    UCXENDPOINT                     *EndpointsToEnable;
    ULONG                           EndpointsToDisableCount;
    UCXENDPOINT                     *EndpointsToDisable;
    ULONG                           EndpointsEnabledAndUnchangedCount;
    UCXENDPOINT                     *EndpointsEnabledAndUnchanged;
    ENDPOINTS_CONFIGURE_FAILURE_FLAGS FailureFlags;
    ULONG                           ExitLatencyDelta;
    UCHAR                           ConfigurationValue;
    UCHAR                           InterfaceNumber;
    UCHAR                           AlternateSetting;
    ULONG                           Reserved1;
    PVOID                           Reserved2;
} ENDPOINTS_CONFIGURE, *PENDPOINTS_CONFIGURE;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that stores handles to the USB hub or device whose endpoints.

EndpointsToEnableCount

The number of endpoints to configure.

EndpointsToEnable

A pointer to the first endpoint handle in the array of endpoints to enable.

EndpointsToDisableCount

The number of endpoints to configure.

`EndpointsToDisable`

A pointer to the first endpoint handle in the array of endpoints to enable.

`EndpointsEnabledAndUnchangedCount`

The number of endpoints that were enabled and unchanged.

`EndpointsEnabledAndUnchanged`

A pointer to the first endpoint handle in the array of endpoints that have not been changed.

`FailureFlags`

The errors, if any, that might occur when attempting to configure endpoints for the USB device or hub.

`ExitLatencyDelta`

The Exit Latency Delta (ELD) value. For more information see section 4.6.6.1 of the eXtensible Host Controller Interface specification.

`ConfigurationValue`

The configuration number of the USB configuration that contains the endpoints.

`InterfaceNumber`

The interface number of the USB interface that contains the endpoints.

`AlternateSetting`

The setting number of the alternate setting that contains the endpoints.

`Reserved1`

`Reserved2`

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#)

# ENDPOINTS\_CONFIGURE\_FAILURE\_FLAG

## S structure (ucxendpoint.h)

Article02/22/2024

This structure provides failure flags to indicate errors, if any, that might have occurred during a request to an [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback function.

## Syntax

C++

```
typedef struct _ENDPOINTS_CONFIGURE_FAILURE_FLAGS {
    ULONG InsufficientBandwidth : 1;
    ULONG InsufficientHardwareResourcesForEndpoints : 1;
    ULONG MaxExitLatencyTooLarge : 1;
    ULONG Reserved : 29;
} ENDPOINTS_CONFIGURE_FAILURE_FLAGS;
```

## Members

`InsufficientBandwidth`

Insufficient bandwidth to configure the specified endpoints.

`InsufficientHardwareResourcesForEndpoints`

Insufficient hardware resources to configure the specified endpoints.

`MaxExitLatencyTooLarge`

The maximum exit latency is too large to configure the specified endpoints.

`Reserved`

Do not use.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[ENDPOINTS\\_CONFIGURE](#)

# EVT\_UCX\_DEFAULT\_ENDPOINT\_UPDATE callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls with information about the default endpoint.

## Syntax

C++

```
EVT_UCX_DEFAULT_ENDPOINT_UPDATE EvtUcxDefaultEndpointUpdate;

void EvtUcxDefaultEndpointUpdate(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `Request`

A [DEFAULT\\_ENDPOINT\\_UPDATE](#) structure that contains the handle to the default endpoint to be updated.

## Return value

None

## Remarks

The UCX client driver registers its *EVT\_UCX\_DEFAULT\_ENDPOINT\_UPDATE* implementation with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

UCX typically calls this routine to update the default endpoint's maximum packet size. The client driver returns completion status in the WDFREQUEST, which it can complete asynchronously.

## Examples

```
VOID
Endpoint_EvtUcxDefaultEndpointUpdate(
    UCXCONTROLLER    UcxController,
    WDFREQUEST       Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, Endpoint, "Endpoint_EvtUcxDefaultEndpointUpdate");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);

    defaultEndpointUpdate =
(PDEFAULT_ENDPOINT_UPDATE)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

## See also

`UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS_INIT`

`UcxDefaultEndpointInitSetEventCallbacks`

# EVT\_UCX\_ENDPOINT\_ABORT callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to abort the queue associated with the endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_ABORT EvtUcxEndpointAbort;

void EvtUcxEndpointAbort(
    [in] UCXCONTROLLER UcxController,
    [in] UCXENDPOINT UcxEndpoint
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxEndpoint

A handle to a UCXENDPOINT object.

## Return value

None

## Remarks

The client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

This function completes all requests associated with the endpoint, typically by calling [WdfIoQueueStopAndPurge](#).

## Examples

C++

```
VOID  
Endpoint_UcxEvtEndpointAbort(  
    UCXCONTROLLER    UcxController,  
    UCXENDPOINT      UcxEndpoint  
)  
{  
    WdfIoQueueStopAndPurge(endpointContext->WdfQueue,  
                           Endpoint_WdfEvtAbortComplete,  
                           UcxEndpoint);  
}
```

## Requirements

  [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# EVT\_UCX\_ENDPOINT\_GET\_ISOCH\_TRAN SFER\_PATH\_DELAYS callback function (ucxendpoint.h)

Article02/22/2024

UCX invokes this callback function to get information about transfer path delays for an isochronous endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_GET_ISOCH_TRANSFER_PATH_DELAYS
EvtUcxEndpointGetIsochTransferPathDelays;

NTSTATUS EvtUcxEndpointGetIsochTransferPathDelays(
    [in]        UCXENDPOINT UcxEndpoint,
    [in, out]   PUCX_ENDPOINT_ISOCH_TRANSFER_PATH_DELAYS
UcxEndpointTransferPathDelays
)
{...}
```

## Parameters

[in] `UcxEndpoint`

A handle to a UCXENDPOINT object that represents the isochronous endpoint for which the client driver receives the transfer path delays.

[in, out] `UcxEndpointTransferPathDelays`

A pointer to a [UCX\\_ENDPOINT\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#) structure that contains transfer path delay values.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

# Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

# Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

- [USB client drivers for Media-Agnostic \(MA-USB\)](#)
- [\\_URB\\_GET\\_ISOCH\\_PIPE\\_TRANSFER\\_PATH\\_DELAYS](#)

# EVT\_UCX\_ENDPOINT\_OK\_TO\_CANCEL\_TRANSFERS callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to notify the controller driver that it can complete cancelled transfers on the endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_OK_TO_CANCEL_TRANSFERS EvtUcxEndpointOkToCancelTransfers;

void EvtUcxEndpointOkToCancelTransfers(
    [in] UCXENDPOINT UcxEndpoint
)
{...}
```

## Parameters

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

Before completing the URB associated with the transfer, the client driver calls [UcxEndpointNeedToCancelTransfers](#) and then waits for UCX to call this function. Then the client driver can complete the URB with **STATUS\_CANCELED**.

 **Note**

If GUID\_USB\_CAPABILITY\_CLEAR\_TT\_BUFFER\_ON\_ASYNC\_TRANSFER\_CANCEL capability is supported, the hub driver may send a control transfer to clear the TT (Transaction Translator) buffer before UCX calls this function.

# Requirements

 [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# EVT\_UCX\_ENDPOINT\_PURGE callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that completes all outstanding I/O requests on the endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_PURGE EvtUcxEndpointPurge;

void EvtUcxEndpointPurge(
    [in] UCXCONTROLLER UcxController,
    [in] UCXENDPOINT UcxEndpoint
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

Typically, this function calls [WdfIoQueuePurge](#).

After UCX calls this function, the client driver fails subsequent I/O requests until UCX calls the client driver's [EVT\\_UCX\\_ENDPOINT\\_START](#) callback function.

## Examples

C++

```
VOID  
Endpoint_UcxEvtEndpointPurge(  
    UCXCONTROLLER    UcxController,  
    UCXENDPOINT     UcxEndpoint  
)  
{  
    WdfIoQueuePurge(endpointContext->WdfQueue,  
                      Endpoint_WdfEvtPurgeComplete,  
                      UcxEndpoint);  
}
```

## Requirements

[+] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

## See also

- [WdfIoQueuePurge](#)

# EVT\_UCX\_ENDPOINT\_RESET callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to reset the controller's programming for an endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_RESET EvtUcxEndpointReset;

void EvtUcxEndpointReset(
    [in] UCXCONTROLLER UcxController,
    [in] UCXENDPOINT UcxEndpoint,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

[in] Request

A handle to a framework request object that the client driver completes when the reset operation is finished.

## Return value

None

# Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

The client driver returns completion status in the WDFREQUEST, which it might complete asynchronously.

## Examples

C++

```
VOID
Endpoint_EvtUcxEndpointReset(
    UCXCONTROLLER    UcxController,
    UCXENDPOINT      UcxEndpoint,
    WDFREQUEST       Request
)

{
    UNREFERENCED_PARAMETER(UcxController);
    UNREFERENCED_PARAMETER(UcxEndpoint);

    DbgTrace(TL_INFO, Endpoint, "Endpoint_EvtUcxEndpointReset");

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

# Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# EVT\_UCX\_ENDPOINT\_SET\_CHARACTERISTIC callback function (ucxendpoint.h)

Article 02/22/2024

UCX invokes this callback function to set the priority on an endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_SET_CHARACTERISTIC EvtUcxEndpointSetCharacteristic;

void EvtUcxEndpointSetCharacteristic(
    [in] UCXENDPOINT UcxEndpoint,
    [in] PUCX_ENDPOINT_CHARACTERISTIC UcxEndpointCharacteristic
)
{...}
```

## Parameters

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

[in] UcxEndpointCharacteristic

A pointer to a [UCX\\_ENDPOINT\\_CHARACTERISTIC](#) structure that contains endpoint characteristics.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

## Requirements

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

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

# EVT\_UCX\_ENDPOINT\_START callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to start the queue associated with the endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_START EvtUcxEndpointStart;

void EvtUcxEndpointStart(
    [in] UCXCONTROLLER UcxController,
    [in] UCXENDPOINT UcxEndpoint
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

# Examples

C++

```
VOID  
Endpoint_EvtUcxEndpointStart(  
    UCXCONTROLLER    UcxController,  
    UCXENDPOINT     UcxEndpoint  
)  
  
{  
    UNREFERENCED_PARAMETER(UcxController);  
    UNREFERENCED_PARAMETER(UcxEndpoint);  
  
    DbgTrace(TL_INFO, Endpoint, "Endpoint_EvtUcxEndpointStart");  
}
```

# Requirements

  Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_ADD callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to create static streams.

## Syntax

C++

```
EVT_UCX_ENDPOINT_STATIC_STREAMS_ADD EvtUcxEndpointStaticStreamsAdd;

NTSTATUS EvtUcxEndpointStaticStreamsAdd(
    [in] UCXENDPOINT UcxEndpoint,
    [in] ULONG NumberOfStreams,
    [in] PUCXSSTREAMS_INIT UcxStaticStreamsInit
)
{...}
```

## Parameters

[in] `UcxEndpoint`

A handle to a UCXENDPOINT object that represents the endpoint.

[in] `NumberOfStreams`

The number of non-default streams to create.

[in] `UcxStaticStreamsInit`

A pointer to an opaque structure containing initialization information. This structure is managed by UCX.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

This callback function creates a UCX static streams object by calling the [UcxStaticStreamsCreate](#) method. Only one UCX static streams object can be associated with a single endpoint. The driver then calls [UcxStaticStreamsSetStreamInfo](#) once per stream to create a queue for each stream.

A static streams object is not enabled until UCX calls the client driver's `EVT_UCX_ENDPOINT_STATIC_STREAMS_ENABLE` callback function.

## Examples

C++

```
NTSTATUS
Endpoint_EvtEndpointStaticStreamsAdd(
    UCXENDPOINT           UcxEndpoint,
    ULONG                 NumberOfStreams,
    PUCXSSTREAMS_INIT     UcxStaticStreamsInit
)
{
    NTSTATUS               status;
    WDF_OBJECT_ATTRIBUTES   wdfAttributes;
    UCXSSTREAMS             ucxStaticStreams;
    STREAM_INFO              streamInfo;
    ULONG                   streamId;

    TRY {

        WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&wdfAttributes,
STATIC_STREAMS_CONTEXT);

        status = UcxStaticStreamsCreate(UcxEndpoint,
                                         &UcxStaticStreamsInit,
                                         &wdfAttributes,
                                         &ucxStaticStreams);
        // ... error handling ...

        for (i = 0, streamId = 1; i < NumberOfStreams; i += 1, streamId +=
1) {

            // ... create WDF queue ...

            STREAM_INFO_INIT(&streamInfo,
                             wdfQueue,
                             streamId):
    }
}
```

```
    UcxStaticStreamsSetStreamInfo(ucxStaticStreams, &streamInfo);  
}
```

# Requirements

[Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	PASSIVE_LEVEL

# EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_DISABLE callback function (ucxendpoint.h)

Article 02/22/2024

The client driver's implementation that UCX calls to release controller resources for all streams for an endpoint.

## Syntax

C++

```
EVT_UCX_ENDPOINT_STATIC_STREAMS_DISABLE EvtUcxEndpointStaticStreamsDisable;

void EvtUcxEndpointStaticStreamsDisable(
    [in] UCXENDPOINT UcxEndpoint,
    [in] UCXSSTREAMS UcxStaticStreams,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

[in] UcxStaticStreams

A handle to a UCX object that represents the static streams.

[in] Request

Contains the URB for the **URB\_FUNCTION\_CLOSE\_STATIC\_STREAMS**.

## Return value

None

# Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

The client driver returns completion status in *Request* and in the USBD\_STATUS in the URB header. The driver can complete the WDFREQUEST asynchronously.

## Examples

C++

```
VOID
Endpoint_EvtUcxEndpointStaticStreamsDisable(
    UCXENDPOINT      UcxEndpoint,
    UCXSSTREAMS      UcxStaticStreams,
    WDFREQUEST       Request
)

{
    UNREFERENCED_PARAMETER(UcxEndpoint);
    UNREFERENCED_PARAMETER(UcxStaticStreams);

    DbgTrace(TL_INFO, Endpoint,
        "Endpoint_EvtUcxEndpointStaticStreamsDisable");

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

# Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# EVT\_UCX\_ENDPOINT\_STATIC\_STREAMS\_ENABLE callback function (ucxendpoint.h)

Article02/22/2024

The client driver's implementation that UCX calls to enable the static streams.

## Syntax

C++

```
EVT_UCX_ENDPOINT_STATIC_STREAMS_ENABLE EvtUcxEndpointStaticStreamsEnable;

void EvtUcxEndpointStaticStreamsEnable(
    [in] UCXENDPOINT UcxEndpoint,
    [in] UCXSSTREAMS UcxStaticStreams,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxEndpoint

A handle to a UCXENDPOINT object that represents the endpoint.

[in] UcxStaticStreams

A handle to a UCX object that represents the static streams.

[in] Request

Contains the URB for the **URB\_FUNCTION\_OPEN\_STATIC\_STREAMS**.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxEndpointCreate](#) method.

The client driver returns completion status in *Request* and in the USBD\_STATUS in the URB header. The driver can complete the WDFREQUEST asynchronously.

## Examples

C++

```
VOID  
Endpoint_EvtUcxEndpointStaticStreamsEnable(  
    UCXENDPOINT    UcxEndpoint,  
    UCXSSTREAMS    UcxStaticStreams,  
    WDFREQUEST     Request  
)  
  
{  
    UNREFERENCED_PARAMETER(UcxEndpoint);  
    UNREFERENCED_PARAMETER(UcxStaticStreams);  
  
    DbgTrace(TL_INFO, Endpoint,  
        "Endpoint_EvtUcxEndpointStaticStreamsEnable");  
  
    WdfRequestComplete(Request, STATUS_SUCCESS);  
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	DISPATCH_LEVEL

# UCX\_CONTROLLER\_ENDPOINT\_CHARACTERISTIC\_PRIORITY enumeration (ucxendpoint.h)

Article 02/22/2024

Indicates the priority of endpoints.

## Syntax

C++

```
typedef enum _UCX_ENDPOINT_CHARACTERISTIC_PRIORITY {
    UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_NONE,
    UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_VIDEO,
    UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_VOICE,
    UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_INTERACTIVE
} UCX_CONTROLLER_ENDPOINT_CHARACTERISTIC_PRIORITY;
```

## Constants

[ ] Expand table

<code>UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_NONE</code> No characteristics of the endpoint.
<code>UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_VIDEO</code> Bulk endpoint with video has the priority.
<code>UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_VOICE</code> Bulk endpoint with voice has the priority.
<code>UCX_ENDPOINT_CHARACTERISTIC_PRIORITY_BULK_INTERACTIVE</code> Bulk endpoint with interactive has the priority.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucxendpoint.h (include Ucxclass.h)

## See also

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

# UCX\_DEFAULT\_ENDPOINT\_EVENT\_CALLBACKS structure (ucxendpoint.h)

Article 02/22/2024

This structure provides a list of UCX default endpoint event callback functions.

## Syntax

C++

```
typedef struct _UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS {
    ULONG                               Size;
    PFN_UCX_ENDPOINT_PURGE             EvtEndpointPurge;
    PFN_UCX_ENDPOINT_START             EvtEndpointStart;
    PFN_UCX_ENDPOINT_ABORT             EvtEndpointAbort;
    PFN_UCX_ENDPOINT_OK_TO_CANCEL_TRANSFERS EvtEndpointOkToCancelTransfers;
    PFN_UCX_DEFAULT_ENDPOINT_UPDATE     EvtDefaultEndpointUpdate;
    HANDLE                             Reserved1;
} UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS,
*PUCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS;
```

## Members

Size

The size in bytes of this structure.

EvtEndpointPurge

A pointer to a EVT\_UCX\_ENDPOINT\_PURGE callback function.

EvtEndpointStart

A pointer to a EVT\_UCX\_ENDPOINT\_START callback function.

EvtEndpointAbort

A pointer to a EVT\_UCX\_ENDPOINT\_ABORT callback function.

EvtEndpointOkToCancelTransfers

A pointer to a EVT\_UCX\_ENDPOINT\_OK\_TO\_CANCEL\_TRANSFERS callback function.

## `EvtDefaultEndpointUpdate`

A pointer to a EVT\_UCX\_DEFAULT\_ENDPOINT\_UPDATE callback function.

### `Reserved1`

Do not use.

## Requirements

[+] Expand table

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

[UcxDefaultEndpointInitSetEventCallbacks](#)

# UCX\_DEFAULT\_ENDPOINT\_EVENT\_CALLBACKS\_INIT function (ucxendpoint.h)

Article 02/22/2024

Initializes a [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS](#) structure with client driver's callback functions. The client driver calls this function before calling [UcxEndpointCreate](#) method to create an endpoint and register its callback functions with UCX.

## Syntax

C++

```
void UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS_INIT(
    [out] PUCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS     Callbacks,
    [in]  PFN_UCX_ENDPOINT_PURGE                   EvtEndpointPurge,
    [in]  PFN_UCX_ENDPOINT_START                  EvtEndpointStart,
    [in]  PFN_UCX_ENDPOINT_ABORT                  EvtEndpointAbort,
    [in]  PFN_UCX_ENDPOINT_OK_TO_CANCEL_TRANSFERS EvtEndpointOkToCancelTransfers,
    [in]  PFN_UCX_DEFAULT_ENDPOINT_UPDATE        EvtDefaultEndpointUpdate
);
```

## Parameters

[out] Callbacks

A pointer to a [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS](#) structure that contains pointers to the client driver's event callback functions.

[in] EvtEndpointPurge

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_PURGE](#) event callback function.

[in] EvtEndpointStart

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_START](#) event callback function.

[in] EvtEndpointAbort

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_ABORT](#) event callback function.

[in] EvtEndpointOkToCancelTransfers

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_OK\\_TO\\_CANCEL\\_TRANSFERS](#) event callback function.

[in] EvtDefaultEndpointUpdate

A pointer to client driver's implementation of the [EVT\\_UCX\\_DEFAULT\\_ENDPOINT\\_UPDATE](#) event callback function.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[UcxEndpointCreate](#)

# UCX\_ENDPOINT\_CHARACTERISTIC structure (ucxendpoint.h)

Article02/22/2024

Stores the characteristics of an endpoint.

## Syntax

C++

```
typedef struct _UCX_ENDPOINT_CHARACTERISTIC {
    ULONG                         Size;
    UCX_ENDPOINT_CHARACTERISTIC_TYPE CharacteristicType;
    union {
        UCX_CONTROLLER_ENDPOINT_CHARACTERISTIC_PRIORITY Priority;
    };
} UCX_ENDPOINT_CHARACTERISTIC, *PUCX_ENDPOINT_CHARACTERISTIC;
```

## Members

Size

Size of this structure.

CharacteristicType

A [UCX\\_ENDPOINT\\_CHARACTERISTIC\\_TYPE](#)-type value that indicates the type of endpoint characteristic.

Priority

A [UCX\\_CONTROLLER\\_ENDPOINT\\_CHARACTERISTIC\\_PRIORITY](#)-typed value that indicates the priority of the endpoint.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Header	ucxendpoint.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_ENDPOINT\\_SET\\_CHARACTERISTIC](#)

[UCX\\_CONTROLLER\\_ENDPOINT\\_CHARACTERISTIC\\_PRIORITY](#)

# UCX\_ENDPOINT\_CHARACTERISTIC\_TYPE enumeration (ucxendpoint.h)

Article02/22/2024

Defines values that indicates the type of endpoint characteristic.

## Syntax

C++

```
typedef enum _UCX_ENDPOINT_CHARACTERISTIC_TYPE {
    UCX_ENDPOINT_CHARACTERISTIC_TYPE_PRIORITY
} UCX_ENDPOINT_CHARACTERISTIC_TYPE;
```

## Constants

[+] Expand table

UCX_ENDPOINT_CHARACTERISTIC_TYPE_PRIORITY
---

The type of characteristic of the endpoint.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucxendpoint.h (include Ucxclass.h)

## See also

[UCX\\_ENDPOINT\\_CHARACTERISTIC](#)

# UCX\_ENDPOINT\_EVENT\_CALLBACKS structure (ucxendpoint.h)

Article02/22/2024

This structure provides a list of pointers to UCX endpoint event callback functions.

## Syntax

C++

```
typedef struct _UCX_ENDPOINT_EVENT_CALLBACKS {
    ULONG                                         Size;
    PFN_UCX_ENDPOINT_PURGE                      EvtEndpointPurge;
    PFN_UCX_ENDPOINT_START                       EvtEndpointStart;
    PFN_UCX_ENDPOINT_ABORT                       EvtEndpointAbort;
    PFN_UCX_ENDPOINT_RESET                       EvtEndpointReset;
    PFN_UCX_ENDPOINT_OK_TO_CANCEL_TRANSFERS     EvtEndpointOkToCancelTransfers;
    PFN_UCX_ENDPOINT_STATIC_STREAMS_ADD          EvtEndpointStaticStreamsAdd;
    PFN_UCX_ENDPOINT_STATIC_STREAMS_ENABLE        EvtEndpointStaticStreamsEnable;
    PFN_UCX_ENDPOINT_STATIC_STREAMS_DISABLE       EvtEndpointStaticStreamsDisable;
    HANDLE                                         Reserved1;
    PFN_UCX_ENDPOINT_GET_ISOCH_TRANSFER_PATH_DELAYS EvtEndpointGetIsochTransferPathDelays;
    PFN_UCX_ENDPOINT_SET_CHARACTERISTIC          EvtEndpointSetCharacteristic;
} UCX_ENDPOINT_EVENT_CALLBACKS, *PUCX_ENDPOINT_EVENT_CALLBACKS;
```

## Members

Size

The size in bytes of the structure.

EvtEndpointPurge

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_PURGE](#) callback function.

EvtEndpointStart

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_START](#) callback function.

`EvtEndpointAbort`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_ABORT](#) callback function.

`EvtEndpointReset`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_RESET](#) callback function.

`EvtEndpointOkToCancelTransfers`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_OK\\_TO\\_CANCEL\\_TRANSFERS](#) callback function.

`EvtEndpointStaticStreamsAdd`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ADD](#) callback function.

`EvtEndpointStaticStreamsEnable`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ENABLE](#) callback function.

`EvtEndpointStaticStreamsDisable`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_DISABLE](#) callback function.

`Reserved1`

Do not use.

`EvtEndpointGetIsochTransferPathDelays`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_GET\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#) callback function.

`EvtEndpointSetCharacteristic`

A pointer to an [EVT\\_UCX\\_ENDPOINT\\_SET\\_CHARACTERISTIC](#) callback function.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)

[UcxEndpointInitSetEventCallbacks](#)

# UCX\_ENDPOINT\_EVENT\_CALLBACKS\_INIT function (ucxendpoint.h)

Article 10/21/2021

Initializes a [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS](#) structure with client driver's callback functions. The client driver calls this function before calling [UcxEndpointCreate](#) method to create an endpoint and register its callback functions with UCX.

## Syntax

C++

```
void UCX_ENDPOINT_EVENT_CALLBACKS_INIT(
    [out] PUCX_ENDPOINT_EVENT_CALLBACKS           Callbacks,
    [in]  PFN_UCX_ENDPOINT_PURGE                 EvtEndpointPurge,
    [in]  PFN_UCX_ENDPOINT_START                EvtEndpointStart,
    [in]  PFN_UCX_ENDPOINT_ABORT                EvtEndpointAbort,
    [in]  PFN_UCX_ENDPOINT_RESET                EvtEndpointReset,
    [in]  PFN_UCX_ENDPOINT_OK_TO_CANCEL_TRANSFERS
        EvtEndpointOkToCancelTransfers,
    [in]  PFN_UCX_ENDPOINT_STATIC_STREAMS_ADD   EvtEndpointStaticStreamsAdd,
    [in]  PFN_UCX_ENDPOINT_STATIC_STREAMS_ENABLE
        EvtEndpointStaticStreamsEnable,
    [in]  PFN_UCX_ENDPOINT_STATIC_STREAMS_DISABLE
        EvtEndpointStaticStreamsDisable
);
```

## Parameters

[out] `Callbacks`

A pointer to a [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS](#) structure that contains pointers to the client driver's event callback functions.

[in] `EvtEndpointPurge`

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_PURGE](#) event callback function.

[in] `EvtEndpointStart`

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_START](#) event callback function.

[in] EvtEndpointAbort

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_ABORT](#) event callback function.

[in] EvtEndpointReset

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_RESET](#) event callback function.

[in] EvtEndpointOkToCancelTransfers

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_OK\\_TO\\_CANCEL\\_TRANSFERS](#) event callback function.

[in] EvtEndpointStaticStreamsAdd

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ADD](#) event callback function.

[in] EvtEndpointStaticStreamsEnable

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ENABLE](#) event callback function.

[in] EvtEndpointStaticStreamsDisable

A pointer to client driver's implementation of the [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_DISABLE](#) event callback function.

## Return value

None

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

**Header**

ucxendpoint.h (include Ucxclass.h)

## See also

[UcxEndpointCreate](#)

# UCX\_ENDPOINT\_ISOCH\_TRANSFER\_PATH\_DELAYS structure (ucxendpoint.h)

Article 02/22/2024

Stores the isochronous transfer path delay values.

## Syntax

C++

```
typedef struct _UCX_ENDPOINT_ISOCH_TRANSFER_PATH_DELAYS {
    ULONG MaximumSendPathDelayInMilliSeconds;
    ULONG MaximumCompletionPathDelayInMilliSeconds;
} UCX_ENDPOINT_ISOCH_TRANSFER_PATH_DELAYS,
*PUCX_ENDPOINT_ISOCH_TRANSFER_PATH_DELAYS;
```

## Members

MaximumSendPathDelayInMilliSeconds

The maximum delay in milliseconds from the time the client driver's isochronous transfer is received by the USB driver stack to the time the transfer is programmed in the host controller. The host controller could either be a local host (as in case of wired USB) or it could be a remote controller as in case of Media-Agnostic USB (MA-USB). In case of MA-USB, it includes the maximum delay associated with the network medium.

MaximumCompletionPathDelayInMilliSeconds

The maximum delay in milliseconds from the time an isochronous transfer is completed by the (local or remote) host controller to the time the corresponding client driver's request is completed by the USB driver stack. For MA-USB, it includes the maximum delay associated with the network medium.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Header	ucxendpoint.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_ENDPOINT\\_GET\\_ISOCH\\_TRANSFER\\_PATH\\_DELAYS](#)

# UcxDefaultEndpointInitSetEventCallbacks function (ucxendpoint.h)

Article02/22/2024

Initializes a **UCXENDPOINT\_INIT** structure with client driver's event callback functions related to the default endpoint.

## Syntax

C++

```
void UcxDefaultEndpointInitSetEventCallbacks(
    PUCXENDPOINT_INIT             EndpointInit,
    PUCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS EventCallbacks
);
```

## Parameters

**EndpointInit**

A pointer to a **UCXENDPOINT\_INIT** structure that UCX passes when it invokes the client driver's [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) event callback function.

**EventCallbacks**

A pointer to a **UCX\_ENDPOINT\_EVENT\_CALLBACKS** structure that contains function pointer to event callback functions related to the endpoint. The client driver initializes the structure by calling [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#).

## Return value

None

## Remarks

The client driver calls this method to set function pointers to its event callback functions just before calling [UcxEndpointCreate](#) to create the default endpoint.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

[UcxEndpointCreate](#)

# UcxEndpointAbortComplete function (ucxendpoint.h)

Article02/22/2024

Notifies UCX that a transfer abort operation has been completed on the specified endpoint object.

## Syntax

C++

```
void UcxEndpointAbortComplete(
    [in] UCXENDPOINT Endpoint
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxEndpointCreate function (ucxendpoint.h)

Article02/22/2024

Creates an endpoint on the specified USB device object.

## Syntax

C++

```
NTSTATUS UcxEndpointCreate(
    [in]          UCXUSBDEVICE           UsbDevice,
    [out]         PUCXENDPOINT_INIT     *EndpointInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]          UCXENDPOINT          *Endpoint
);
```

## Parameters

[in] UsbDevice

A handle to the USB device object that contains the endpoint. The client driver retrieved the handle in a previous call to [UcxUsbDeviceCreate](#).

[out] EndpointInit

A pointer to a **UCXENDPOINT\_INIT** structure that describes various configuration operations for creating the endpoint object. The driver specifies function pointers to its callback functions in this structure. This structure is managed by UCX.

[in, optional] Attributes

A pointer to a caller-allocated **WDF\_OBJECT\_ATTRIBUTES** structure that specifies attributes for the endpoint object.

[out] Endpoint

A pointer to a variable that receives a handle to the new endpoint object.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return one an appropriate NTSTATUS error code.

## Remarks

The client driver for the host controller must call this method after the [WdfDeviceCreate](#) call. The parent of the new endpoint object is the USB device object.

The method initializes the endpoint object with information such as the type of endpoint, pipe, transfer, and maximum transfers size.

For a code example, see [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	PASSIVE_LEVEL

# UcxEndpointGetStaticStreamsReferenced function (ucxendpoint.h)

Article 10/21/2021

Returns a referenced static streams object for the specified endpoint.

## Syntax

C++

```
UCXSSTREAMS UcxEndpointGetStaticStreamsReferenced(
    [in] UCXENDPOINT Endpoint,
    [in] PVOID      Tag
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object for which the static streams object is requested. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

[in] Tag

A driver-defined value that the framework stores as an identification tag for the object reference.

## Return value

A handle to the stream object if it is opened with the endpoint. Otherwise, NULL.

## Remarks

The client driver can use this function to determine whether it has created a streams object for this endpoint. If it creates the object the method returns the UCXSSTREAMS handle. The method returns NULL if the object was not created, or if the client driver failed the framework request object passed in the

[EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ENABLE](#) callback. Any call to this method must be matched by a call to [WdfObjectDereferenceWithTag](#) by using the same Tag.

# Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxEndpointInitSetEventCallbacks function (ucxendpoint.h)

Article 02/22/2024

Initializes a **UCXENDPOINT\_INIT** structure with client driver's event callback functions related to endpoints on the device.

## Syntax

C++

```
void UcxEndpointInitSetEventCallbacks(
    PUCXENDPOINT_INIT          EndpointInit,
    PUCX_ENDPOINT_EVENT_CALLBACKS EventCallbacks
);
```

## Parameters

**EndpointInit**

A pointer to a **UCXENDPOINT\_INIT** structure that UCX passes when it invokes the client driver's [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#) event callback function.

**EventCallbacks**

A pointer to a **UCX\_ENDPOINT\_EVENT\_CALLBACKS** structure that contains function pointer to event callback functions related to the endpoint. The client driver initializes the structure by calling [UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#).

## Return value

None

## Remarks

The client driver calls this method to set function pointers to its event callback functions just before calling [UcxEndpointCreate](#) to create an endpoint.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[UCX\\_ENDPOINT\\_EVENT\\_CALLBACKS](#)

[UcxEndpointCreate](#)

# UcxEndpointNeedToCancelTransfers function (ucxendpoint.h)

Article02/22/2024

The client driver calls this method before it cancels transfers on the wire.

## Syntax

C++

```
void UcxEndpointNeedToCancelTransfers(
    [in] UCXENDPOINT Endpoint
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

## Return value

None

## Remarks

If needed, UCX coordinates with the hub driver to send a Clear TT buffer command to the TT Hub.

After that operation completes, UCX invokes the client driver's EVT\_UCX\_ENDPOINT\_OK\_TO\_CANCEL\_TRANSFERS callback function.

## Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxEndpointNoPingResponseError function (ucxendpoint.h)

Article02/22/2024

Notifies UCX about a "No Ping Response" error for a transfer on the specified endpoint object

## Syntax

C++

```
void UcxEndpointNoPingResponseError(
    [in] UCXENDPOINT Endpoint
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxEndpointPurgeComplete function (ucxendpoint.h)

Article02/22/2024

Notifies UCX that a purge operation has been completed on the specified endpoint object.

## Syntax

C++

```
void UcxEndpointPurgeComplete(
    [in] UCXENDPOINT Endpoint
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

Requirement	Value
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxEndpointCreate](#)

# UcxEndpointSetWdfIoQueue function (ucxendpoint.h)

Article02/22/2024

Sets a framework queue on the specified endpoint object.

## Syntax

C++

```
void UcxEndpointSetWdfIoQueue(
    [in] UCXENDPOINT Endpoint,
    [in] WDFQUEUE     WdfQueue
);
```

## Parameters

[in] Endpoint

A handle to the endpoint object. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

[in] WdfQueue

A handle to the framework queue object to set on the endpoint.

## Return value

None

## Remarks

This routine can only get called from [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#) and [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) callback functions. The client driver must call this routine only once for each endpoint.

For a code example, see [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#).

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxendpoint.h (include Ucxclass.h, Ucxendpoint.h)

## See also

[UcxEndpointCreate](#)

# ucxroothub.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucxroothub.h contains the following programming interfaces:

## Functions

### [UCX\\_ROOTHUB\\_CONFIG\\_INIT\\_WITH\\_CONTROL\\_URB\\_HANDLER](#)

Learn more about the UCX\_ROOTHUB\_CONFIG\_INIT\_WITH\_CONTROL\_URB\_HANDLER function.

### [UcxRootHubCreate](#)

Learn more about the UcxRootHubCreate function.

### [UcxRootHubPortChanged](#)

Notifies UCX about a new port change event on the host controller.

## Callback functions

### [EVT\\_UCX\\_ROOTHUB\\_CONTROL\\_URB](#)

The client driver uses this callback type to implement handlers that UCX calls when it receives feature control requests on the USB hub.

### [EVT\\_UCX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about USB 2.0 ports on the root hub.

### [EVT\\_UCX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about USB 3.0 ports on the root hub.

## [EVT\\_UCX\\_ROOTHUB\\_GET\\_INFO](#)

The client driver's implementation that UCX calls when it receives a request for information about the root hub.

## [EVT\\_UCX\\_ROOTHUB\\_INTERRUPT\\_TX](#)

The client driver's implementation that UCX calls when it receives a request for information about changed ports.

# Structures

## [CONTROLLER\\_USB\\_20\\_HARDWARE\\_LPM\\_FLAGS](#)

Describes supported protocol capabilities for Link Power Management (LPM) in as defined the USB 2.0 specification.

## [HUB\\_INFO\\_FROM\\_PARENT](#)

Describes information about a hub from its parent device.

## [PARENT\\_HUB\\_FLAGS](#)

This structure is used by the HUB\_INFO\_FROM\_PARENT structure to get hub information from the parent.

## [ROOTHUB\\_20PORT\\_INFO](#)

Provides information about a USB 2.0 root hub port. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_20PORT\_INFO callback function.

## [ROOTHUB\\_20PORTS\\_INFO](#)

This structure that has an array of 2.0 ports supported by the root hub. This structure is provided by UCX in a framework request in the EVT\_UCX\_ROOTHUB\_GET\_20PORT\_INFO callback function.

## [ROOTHUB\\_30PORT\\_INFO](#)

Provides information about a USB 3.0 root hub port. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_30PORT\_INFO callback function.

## [ROOTHUB\\_30PORT\\_INFO\\_EX](#)

Provides extended USB 3.0 port information about speed.

### [ROOTHUB\\_30PORTS\\_INFO](#)

Provides information about USB 3.0 root hub ports. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_30PORT\_INFO callback function.

### [ROOTHUB\\_INFO](#)

Provides information about a USB root hub. This structure is passed by UCX in the EVT\_UCX\_ROOTHUB\_GET\_INFO callback function.

### [UCX\\_ROOTHUB\\_CONFIG](#)

Contains pointers to event callback functions for creating the root hub by calling UcxRootHubCreate. Initialize this structure by calling UCX\_ROOTHUB\_CONFIG\_INIT initialization function (see Ucxclass.h).

## Enumerations

### [CONTROLLER\\_TYPE](#)

This enumeration specifies if the USB host controller is an eXtensible Host Controller Interface (xHCI) controller.

### [TRISTATE](#)

The TRISTATE enumeration indicates generic state values for true or false.

# CONTROLLER\_TYPE enumeration (ucxroothub.h)

Article02/22/2024

This enumeration specifies if the USB host controller is an eXtensible Host Controller Interface (xHCI) controller.

## Syntax

C++

```
typedef enum _CONTROLLER_TYPE {
    ControllerTypeXhci,
    ControllerTypeSoftXhci
} CONTROLLER_TYPE;
```

## Constants

[ ] Expand table

<code>ControllerTypeXhci</code>	Indicates the USB host controller is an xHCI controller.
<code>ControllerTypeSoftXhci</code>	Indicates the USB host controller is software an xHCI controller.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

[ROOTHUB\\_INFO](#)

# CONTROLLER\_USB\_20\_HARDWARE\_LPM\_FLAGS union (ucxroothub.h)

Article 02/22/2024

Describes supported protocol capabilities for Link Power Management (LPM) in as defined the USB 2.0 specification.

## Syntax

C++

```
typedef union _CONTROLLER_USB_20_HARDWARE_LPM_FLAGS {
    UCHAR AsUchar;
    struct {
        UCHAR L1CapabilitySupported : 1;
        UCHAR BeslLpmCapabilitySupported : 1;
    } Flags;
    struct {
        UCHAR L1CapabilitySupported : 1;
        UCHAR BeslLpmCapabilitySupported : 1;
    };
} CONTROLLER_USB_20_HARDWARE_LPM_FLAGS,
*PCONTROLLER_USB_20_HARDWARE_LPM_FLAGS;
```

## Members

AsUchar

The size of structure represented as a char (8-bit) value.

Flags

Flags.L1CapabilitySupported

Flags.BeslLpmCapabilitySupported

L1CapabilitySupported

Indicates support for L1 transitions.

BeslLpmCapabilitySupported

Indicates Best Effort Service latency (BESL) latency support.

# Requirements

 Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

[ROOTHUB\\_20PORT\\_INFO](#)

# EVT\_UCX\_ROOTHUB\_CONTROL\_URB callback function (ucxroothub.h)

Article 10/21/2021

The client driver uses this callback type to implement handlers that UCX calls when it receives feature control requests on the USB hub.

## Syntax

C++

```
EVT_UCX_ROOTHUB_CONTROL_URB EvtUcxRoothubControlUrb;

void EvtUcxRoothubControlUrb(
    [in] UCXROOTHUB UcxRootHub,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxRootHub`

A handle to a UCX object that represents the root hub.

[in] `Request`

Contains the [URB](#) for the feature request.

## Return value

None

## Remarks

The client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxRootHubCreate](#) method.

The driver either provides callbacks for all of the individual feature request types, as shown in the first example, or it can provide a single handler of type

*EVT\_UCX\_ROOTHUB\_CONTROL\_URB* that UCX calls for all hub and port control transfers.

The client driver returns completion status in *Request* and in the USBD\_STATUS in the URB header. The driver can complete the WDFREQUEST asynchronously.

## Examples

This example shows how to register callbacks for individual feature request types.

```
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubClearHubFeature;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubClearPortFeature;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubGetHubStatus;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubGetPortStatus;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubSetHubFeature;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubSetPortFeature;
EVT_UCX_ROOTHUB_CONTROL_URB RootHub_EvtRootHubGetPortErrorCount;

...

// Create the root hub
//
UCX_ROOTHUB_CONFIG_INIT(&ucxRootHubConfig,
                        RootHub_EvtRootHubClearHubFeature,
                        RootHub_EvtRootHubClearPortFeature,
                        RootHub_EvtRootHubGetHubStatus,
                        RootHub_EvtRootHubGetPortStatus,
                        RootHub_EvtRootHubSetHubFeature,
                        RootHub_EvtRootHubSetPortFeature,
                        RootHub_EvtRootHubGetPortErrorCount,
                        RootHub_EvtRootHubInterruptTx,
                        RootHub_EvtRootHubGetInfo,
                        RootHub_EvtRootHubGet20PortInfo,
                        RootHub_EvtRootHubGet30PortInfo);

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes,
                                      UCX_ROOTHUB_CONTEXT);

status = UcxRootHubCreate(ucxController,
                         &ucxRootHubConfig,
                         &objectAttributes,
                         &ucxRootHub);
```

Here is a sample implementation of one of the URB-specific request handlers.

```

VOID
RootHub_EvtRootHubClearHubFeature(
    UCXROOTHUB           UcxRootHub,
    WDFREQUEST            ControlUrb
)
/*++

Routine Description:

    UCX calls this routine when it receives a new Clear Hub Feature request.

--*/
{
    UNREFERENCED_PARAMETER(UcxRootHub);

    DbgTrace(TL_INFO, RootHub, "RootHub_EvtRootHubClearHubFeature");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);
    urb = (PURB)wdfRequestParams.Parameters.Others.Arg1;
    setupPacket = (PWDF_USB_CONTROL_SETUP_PACKET)&urb-
        >UrbControlTransferEx.SetupPacket[0];
    ...

    WdfRequestComplete(ControlUrb, STATUS_SUCCESS);
}

```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxroothub.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[UcxRootHubCreate](#)

# EVT\_UCX\_ROOTHUB\_GET\_20PORT\_INFO callback function (ucxroothub.h)

Article02/22/2024

The client driver's implementation that UCX calls when it receives a request for information about USB 2.0 ports on the root hub.

## Syntax

C++

```
EVT_UCX_ROOTHUB_GET_20PORT_INFO EvtUcxRoothubGet20PortInfo;

void EvtUcxRoothubGet20PortInfo(
    [in] UCXROOTHUB UcxRootHub,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxRootHub

A handle to a UCX object that represents the root hub.

[in] Request

A structure of type [\\_ROOTHUB\\_20PORT\\_INFO](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxRootHubCreate](#) method.

The **PortInfoArray** array of the [\\_ROOTHUB\\_20PORTS\\_INFO](#) structure contains a list of USB 2.0 ports that the root hub supports.

The client driver returns completion status in *Request* and in the USBD\_STATUS in the URB header. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
RootHub_EvtRootHubGet20PortInfo(
    UCXROOTHUB           UcxRootHub,
    WDFREQUEST           Request
)
/*++

For sample demonstration purposes, this function returns statically
defined information for the single 2.0 port.

--*/
{
    PUCX_ROOTHUB_CONTEXT    ucxRootHubContext;
    WDF_REQUEST_PARAMETERS  wdfRequestParams;
    PROOTHUB_20PORTS_INFO   rootHub20PortsInfo;
    NTSTATUS                 status;

    ucxRootHubContext = GetUcxRootHubContext(UcxRootHub);

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(Request, &wdfRequestParams);

    rootHub20PortsInfo =
    (PROOTHUB_20PORTS_INFO)wdfRequestParams.Parameters.Others.Arg1;

    if (rootHub20PortsInfo->Size < sizeof(ROOTHUB_20PORTS_INFO)) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_20PORTS_INFO Size %d",
rootHub20PortsInfo->Size);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet20PortInfoEnd;
    }

    if (rootHub20PortsInfo->NumberOfPorts != ucxRootHubContext-
>NumberOf20Ports) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_20PORTS_INFO
NumberOfPorts %d", rootHub20PortsInfo->NumberOfPorts);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet20PortInfoEnd;
    }

    if (rootHub20PortsInfo->PortInfoSize < sizeof(ROOTHUB_20PORT_INFO)) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_20PORT_INFO Size %d",
rootHub20PortsInfo->PortInfoSize);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet20PortInfoEnd;
    }
```

```

}

// 
// Return static root hub 2.0 port information.
//
rootHub20PortsInfo->PortInfoArray[0]->PortNumber =
ROOTHUB_20_PORT_PORT_NUMBER;
rootHub20PortsInfo->PortInfoArray[0]->Removable = TriStateTrue;

status = STATUS_SUCCESS;

DbgTrace(TL_INFO, RootHub, "RootHub_EvtRootHubGet20PortInfo");

RootHub_EvtRootHubGet20PortInfoEnd:

WdfRequestComplete(Request, status);
}

```

## Requirements

[] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxroothub.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[\\_ROTHUB\\_20PORTS\\_INFO](#)

# EVT\_UCX\_ROOTHUB\_GET\_30PORT\_INFO callback function (ucxroothub.h)

Article 02/22/2024

The client driver's implementation that UCX calls when it receives a request for information about USB 3.0 ports on the root hub.

## Syntax

C++

```
EVT_UCX_ROOTHUB_GET_30PORT_INFO EvtUcxRoothubGet30PortInfo;

void EvtUcxRoothubGet30PortInfo(
    [in] UCXROOTHUB UcxRootHub,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxRootHub

A handle to a UCX object that represents the root hub.

[in] Request

A structure of type [\\_ROOTHUB\\_30PORT\\_INFO](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxRootHubCreate](#) method.

The **PortInfoArray** array of the [\\_ROOTHUB\\_30PORT\\_INFO](#) structure contains a list of USB 3.0 ports that the root hub supports.

The client driver returns completion status in *Request* and in the USBD\_STATUS in the URB header. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
RootHub_EvtRootHubGet30PortInfo(
    UCXROOTHUB           UcxRootHub,
    WDFREQUEST           Request
)
/*++

For sample demonstration purposes, this function returns statically
defined information for the single 3.0 port.

--*/
{
    PUCX_ROOTHUB_CONTEXT    ucxRootHubContext;
    WDF_REQUEST_PARAMETERS  wdfRequestParams;
    PROOTHUB_30PORTS_INFO   rootHub30PortsInfo;
    NTSTATUS                 status;

    ucxRootHubContext = GetUcxRootHubContext(UcxRootHub);

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(Request, &wdfRequestParams);

    rootHub30PortsInfo =
    (PROOTHUB_30PORTS_INFO)wdfRequestParams.Parameters.Others.Arg1;

    if (rootHub30PortsInfo->Size < sizeof(ROOTHUB_30PORTS_INFO)) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_30PORTS_INFO Size %d",
rootHub30PortsInfo->Size);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet30PortInfoEnd;
    }

    if (rootHub30PortsInfo->NumberOfPorts != ucxRootHubContext-
>NumberOf30Ports) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_30PORTS_INFO
NumberOfPorts %d", rootHub30PortsInfo->NumberOfPorts);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet30PortInfoEnd;
    }

    if (rootHub30PortsInfo->PortInfoSize < sizeof(ROOTHUB_30PORT_INFO)) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_30PORT_INFO Size %d",
rootHub30PortsInfo->PortInfoSize);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGet30PortInfoEnd;
    }
```

```

}

// 
// Return static root hub 3.0 port information.
//
rootHub30PortsInfo->PortInfoArray[0]->PortNumber =
ROOTHUB_30_PORT_PORT_NUMBER;
rootHub30PortsInfo->PortInfoArray[0]->Removable = TriStateTrue;

status = STATUS_SUCCESS;

DbgTrace(TL_INFO, RootHub, "RootHub_EvtRootHubGet30PortInfo");

RootHub_EvtRootHubGet30PortInfoEnd:

WdfRequestComplete(Request, status);
}

```

## Requirements

[] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxroothub.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[\\_ROTHUB\\_30PORT\\_INFO](#)

# EVT\_UCX\_ROOTHUB\_GET\_INFO callback function (ucxroothub.h)

Article 10/21/2021

The client driver's implementation that UCX calls when it receives a request for information about the root hub.

## Syntax

C++

```
EVT_UCX_ROOTHUB_GET_INFO EvtUcxRoothubGetInfo;

void EvtUcxRoothubGetInfo(
    [in] UCXROOTHUB UcxRootHub,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxRootHub`

A handle to a UCX object that represents the root hub.

[in] `Request`

A structure of type [ROTHUB\\_INFO](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxRootHubCreate](#) method.

The [\\_ROTHUB\\_INFO](#) structure contains the number of USB 2.0 and USB 3.0 ports supported by the root hub.

After UCX calls the *EVT\_UCX\_ROOTHUB\_GET\_INFO* function, the number of ports exposed by the root hub is guaranteed to remain the same. Note that these are virtual ports, not physical ports. Each physical USB connector is represented by one or more ports of different speed on the root hub.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
RootHub_EvtRootHubGetInfo(
    UCXROOTHUB        UcxRootHub,
    WDFREQUEST        Request
)
/*++

For sample demonstration purposes, this function returns statically
defined information for the root hub.

--*/
{
    PUCX_ROOTHUB_CONTEXT    ucxRootHubContext;
    WDF_REQUEST_PARAMETERS  wdfRequestParams;
    PROOTHUB_INFO           rootHubInfo;
    NTSTATUS                 status;

    ucxRootHubContext = GetUcxRootHubContext(UcxRootHub);

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(Request, &wdfRequestParams);

    rootHubInfo = (PROOTHUB_INFO)wdfRequestParams.Parameters.Others.Arg1;

    if (rootHubInfo->Size < sizeof(ROOTHUB_INFO)) {
        DbgTrace(TL_ERROR, RootHub, "Invalid ROOTHUB_INFO Size %d",
rootHubInfo->Size);
        status = STATUS_INVALID_PARAMETER;
        goto RootHub_EvtRootHubGetInfo;
    }

    rootHubInfo->ControllerType = ControllerTypeSoftXhci;
    rootHubInfo->NumberOf20Ports = ucxRootHubContext->NumberOf20Ports;
    rootHubInfo->NumberOf30Ports = ucxRootHubContext->NumberOf30Ports;
    rootHubInfo->MaxU1ExitLatency = ucxRootHubContext->U1DeviceExitLatency;
    rootHubInfo->MaxU2ExitLatency = ucxRootHubContext->U2DeviceExitLatency;

    DbgTrace(TL_INFO, RootHub, "RootHub_UcxEvtGetInfo NumberOf20Ports %d
NumberOf30Ports %d", rootHubInfo->NumberOf20Ports, rootHubInfo-
```

```
>NumberOf30Ports);  
  
    status = STATUS_SUCCESS;  
  
RootHub_EvtRootHubGetInfo:  
  
    WdfRequestComplete(Request, status);  
}
```

## Requirements

<b>Target Platform</b>	Windows
<b>Minimum KMDF version</b>	1.0
<b>Minimum UMDF version</b>	2.0
<b>Header</b>	ucxroothub.h (include Ucxclass.h)
<b>IRQL</b>	DISPATCH_LEVEL

## See also

[\\_ROOTHUB\\_INFO](#)

# EVT\_UCX\_ROOTHUB\_INTERRUPT\_TX callback function (ucxroothub.h)

Article 02/22/2024

The client driver's implementation that UCX calls when it receives a request for information about changed ports.

## Syntax

C++

```
EVT_UCX_ROOTHUB_INTERRUPT_TX EvtUcxRoothubInterruptTx;

void EvtUcxRoothubInterruptTx(
    [in] UCXROOTHUB UcxRootHub,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxRootHub`

A handle to a UCX object that represents the root hub.

[in] `Request`

Contains the [URB](#) for the root hub interrupt transfer request.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxRootHubCreate](#) method.

The *Request* parameter contains a buffer in which each bit corresponds to a root hub port, with the first bit corresponding to the first port. The client driver sets the

corresponding bit if any port has changed, and then completes the request.

The client driver returns completion status in *Request*.

## Examples

This snippet shows how the callback extracts the root hub interrupt transfer request.

```
WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
WdfRequestGetParameters(WdfRequest, &wdfRequestParams);

urb = (PURB)wdfRequestParams.Parameters.Others.Arg1;
transferBuffer = urb->UrbBulkOrInterruptTransfer.TransferBuffer;
transferBufferLength = urb-
>UrbBulkOrInterruptTransfer.TransferBufferLength;
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxroothub.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

# HUB\_INFO\_FROM\_PARENT structure (ucxroothub.h)

Article 05/23/2024

Describes information about a hub from its parent device.

## Syntax

C++

```
typedef struct _HUB_INFO_FROM_PARENT {
    PDEVICE_OBJECT IoTarget;
    USB_DEVICE_DESCRIPTOR DeviceDescriptor;
    USHORT U1ExitLatency;
    USHORT U2ExitLatency;
    USHORT ExitLatencyOfSlowestLinkForU1;
    UCHAR DepthOfSlowestLinkForU1;
    USHORT ExitLatencyOfSlowestLinkForU2;
    UCHAR DepthOfSlowestLinkForU2;
    USHORT HostInitiatedU1ExitLatency;
    USHORT HostInitiatedU2ExitLatency;
    UCHAR TotalHubDepth;
    USHORT TotalTPPropogationDelay;
    PARENT_HUB_FLAGS HubFlags;
    PUSB_DEVICE_CAPABILITY_SUPERSPEEDPLUS_SPEED SublinkSpeedAttr;
    ULONG SublinkSpeedAttrCount;
} HUB_INFO_FROM_PARENT, *PHUB_INFO_FROM_PARENT;
```

## Members

`IoTarget`

A pointer to the WDM device object of the parent that represents the I/O target.

`DeviceDescriptor`

A [USB\\_DEVICE\\_DESCRIPTOR](#) structure that contains the device descriptor.

`U1ExitLatency`

The time to transition from the U1 state.

`U2ExitLatency`

The time to transition from the U2 state.

`ExitLatencyOfSlowestLinkForU1`

The exit latency for the slowest link for U1 transition.

`DepthOfSlowestLinkForU1`

The depth of the hub based on which the latency for the slowest link is calculated for a U1 transition.

`ExitLatencyOfSlowestLinkForU2`

The exit latency for the slowest link for U2 transition.

`DepthOfSlowestLinkForU2`

The depth of the hub based on which the latency for the slowest link is calculated for a U2 transition.

`HostInitiatedU1ExitLatency`

Host-initiated exit latency to transition from the U1 state.

`HostInitiatedU2ExitLatency`

Host-initiated exit latency to transition from the U2 state.

`TotalHubDepth`

Total hub depth.

`TotalTPPropogationDelay`

The total TP propagation delay.

`HubFlags`

A bitwise-OR of [PARENT\\_HUB\\_FLAGS](#) flags.

`SublinkSpeedAttr`

A pointer to a **USB\_DEVICE\_CAPABILITY\_SUPERSPEEDPLUS\_SPEED** structure that describes the USB 3.1 capability's sublink speed attributes. For structure declaration, see Usbspec.h

`SublinkSpeedAttrCount`

The count of sublink speed attributes.

# Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# PARENT\_HUB\_FLAGS union (ucxroothub.h)

Article 02/22/2024

This structure is used by the [HUB\\_INFO\\_FROM\\_PARENT](#) structure to get hub information from the parent.

## Syntax

C++

```
typedef union _PARENT_HUB_FLAGS {
    ULONG AsUlong32;
    struct {
        ULONG DisableLpmForAllDownstreamDevices : 1;
        ULONG HubIsHighSpeedCapable : 1;
        ULONG DisableUpdateMaxExitLatency : 1;
        ULONG DisableU1 : 1;
    } Flags;
    struct {
        ULONG DisableLpmForAllDownstreamDevices : 1;
        ULONG HubIsHighSpeedCapable : 1;
        ULONG DisableUpdateMaxExitLatency : 1;
        ULONG DisableU1 : 1;
    };
} PARENT_HUB_FLAGS, *PPARENT_HUB_FLAGS;
```

## Members

AsUlong32

The size of structure represented as a LONG (32-bit) value.

Flags

Flags.DisableLpmForAllDownstreamDevices

Indicates that LPM should be disabled for all devices/hubs behind the this hub.

Flags.HubIsHighSpeedCapable

Indicates that the hub is high-speed capable.

Flags.DisableUpdateMaxExitLatency

Indicates that UpdateMaxExitLatency should be disabled.

`Flags.DisableU1`

Indicates that U1 transitions should be disabled.

`DisableLpmForAllDownstreamDevices`

Indicates that LPM should be disabled for all devices/hubs behind the this hub.

`HubIsHighSpeedCapable`

Indicates that the hub is high-speed capable.

`DisableUpdateMaxExitLatency`

Indicates that UpdateMaxExitLatency should be disabled.

`DisableU1`

Indicates that U1 transitions should be disabled.

## Requirements

[] [Expand table](#)

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

- [HUB\\_INFO\\_FROM\\_PARENT](#)

# ROOTHUB\_20PORT\_INFO structure (ucxroothub.h)

Article 02/22/2024

Provides information about a USB 2.0 root hub port. This structure is passed by UCX in the [EVT\\_UCX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _ROOTHUB_20PORT_INFO {
    USHORT PortNumber;
    UCHAR MinorRevision;
    UCHAR HubDepth;
    TRISTATE Removable;
    TRISTATE IntegratedHubImplemented;
    TRISTATE DebugCapable;
    CONTROLLER_USB_20_HARDWARE_LPM_FLAGS ControllerUsb20HardwareLpmFlags;
} ROOTHUB_20PORT_INFO, *PROOTHUB_20PORT_INFO;
```

## Members

PortNumber

The USB 2.0 root hub port number.

MinorRevision

Minor revision number.

HubDepth

The hub depth limit.

Removable

A [TRISTATE](#) value that indicates if the port is removable.

IntegratedHubImplemented

A [TRISTATE](#) value that indicates if the port is implemented.

`DebugCapable`

A [TRISTATE](#) value that indicates if the port is debug capable.

`ControllerUsb20HardwareLpmFlags`

A value that indicates Link Power Management (LPM) flags for the controller.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

# ROOTHUB\_20PORTS\_INFO structure (ucxroothub.h)

Article 02/22/2024

This structure that has an array of 2.0 ports supported by the root hub. This structure is provided by UCX in a framework request in the [EVT\\_UCX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _ROOTHUB_20PORTS_INFO {
    ULONG             Size;
    USHORT            NumberOfPorts;
    USHORT            PortInfoSize;
    PROOTHUB_20PORT_INFO *PortInfoArray;
} ROOTHUB_20PORTS_INFO, *PROOTHUB_20PORTS_INFO;
```

## Members

Size

The size in bytes of this structure.

NumberOfPorts

The number of USB 2.0 ports connected to the root hub.

PortInfoSize

The size of the ROOTHUB\_20PORTS\_INFO structure.

PortInfoArray

A pointer to an array of PROOTHUB\_20PORT\_INFO structures.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#)

# ROOTHUB\_30PORT\_INFO structure (ucxroothub.h)

Article 02/22/2024

Provides information about a USB 3.0 root hub port. This structure is passed by UCX in the [EVT\\_UCX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _ROOTHUB_30PORT_INFO {
    USHORT    PortNumber;
    UCHAR     MinorRevision;
    UCHAR     HubDepth;
    TRISTATE  Removable;
    TRISTATE  DebugCapable;
} ROOTHUB_30PORT_INFO, *PROOTHUB_30PORT_INFO;
```

## Members

**PortNumber**

The USB 3.0 port number connected to the root hub.

**MinorRevision**

Revision number.

**HubDepth**

The hub depth limit.

**Removable**

A [TRISTATE](#) value that indicates if the port is removable.

**DebugCapable**

A [TRISTATE](#) value that indicates if the port is debug capable.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

# ROOTHUB\_30PORT\_INFO\_EX structure (ucxroothub.h)

Article 02/22/2024

Provides extended USB 3.0 port information about speed.

## Syntax

C++

```
typedef struct _ROOTHUB_30PORT_INFO_EX {
    ROOTHUB_30PORT_INFO           Info;
    USHORT                         MaxSpeedsCount;
    USHORT                         SpeedsCount;
    PUSB_DEVICE_CAPABILITY_SUPERSPEEDPLUS_SPEED Speeds;
} ROOTHUB_30PORT_INFO_EX, *PROOTHUB_30PORT_INFO_EX;
```

## Members

Info

A [ROOTHUB\\_30PORT\\_INFO](#) structure.

MaxSpeedsCount

Maximum number of speeds.

SpeedsCount

The count of bus speeds supported.

Speeds

A pointer to a [USB\\_DEVICE\\_CAPABILITY\\_SUPERSPEEDPLUS\\_SPEED](#) structure that describes the USB 3.1 capability's sublink speed attributes. For structure declaration, see Usbspec.h

## Requirements

[+] [Expand table](#)

<b>Requirement</b>	<b>Value</b>
Header	ucxroothub.h (include Ucxclass.h)

# ROOTHUB\_30PORTS\_INFO structure (ucxroothub.h)

Article02/22/2024

Provides information about USB 3.0 root hub ports. This structure is passed by UCX in the [EVT\\_UCX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _ROOTHUB_30PORTS_INFO {
    ULONG             Size;
    USHORT            NumberOfPorts;
    USHORT            PortInfoSize;
    PROOTHUB_30PORT_INFO *PortInfoArray;
} ROOTHUB_30PORTS_INFO, *PROOTHUB_30PORTS_INFO;
```

## Members

**Size**

The size in bytes of this structure.

**NumberOfPorts**

Number of USB 3.0 root hub ports.

**PortInfoSize**

The size of the [ROOTHUB\\_30PORT\\_INFO](#) array.

**PortInfoArray**

A pointer to an array of [ROOTHUB\\_30PORT\\_INFO](#) structures.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#)

# ROOTHUB\_INFO structure (ucxroothub.h)

Article02/22/2024

Provides information about a USB root hub. This structure is passed by UCX in the [EVT\\_UCX\\_ROOTHUB\\_GET\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _ROOTHUB_INFO {
    ULONG          Size;
    CONTROLLER_TYPE ControllerType;
    USHORT         NumberOf20Ports;
    USHORT         NumberOf30Ports;
    USHORT         MaxU1ExitLatency;
    USHORT         MaxU2ExitLatency;
} ROOTHUB_INFO, *PROOTHUB_INFO;
```

## Members

Size

The size in bytes of this structure.

ControllerType

A [CONTROLLER\\_TYPE](#) value that identifies the type of eXtensible Host Controller Interface (xHCI) which has the root hub.

NumberOf20Ports

The number of USB 2.0 ports connected to the root hub.

NumberOf30Ports

The number of USB 3.0 ports connected to the root hub.

MaxU1ExitLatency

The exit latency for the slowest link for U1 transition.

## MaxU2ExitLatency

The exit latency for the slowest link for U2 transition.

# Requirements

[ ] Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_ROOTHUB\\_GET\\_INFO](#)

# TRISTATE enumeration (ucxroothub.h)

Article 02/22/2024

The **TRISTATE** enumeration indicates generic state values for true or false.

## Syntax

C++

```
typedef enum _TRISTATE {
    TriStateUnknown,
    TriStateFalse,
    TriStateTrue
} TRISTATE;
```

## Constants

[ ] Expand table

<code>TriStateUnknown</code>	State is unknown.
<code>TriStateFalse</code>	State is a false boolean value.
<code>TriStateTrue</code>	State is a true boolean value.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

# UCX\_ROOTHUB\_CONFIG structure (ucxroothub.h)

Article02/22/2024

Contains pointers to event callback functions for creating the root hub by calling [UcxRootHubCreate](#). Initialize this structure by calling [UCX\\_ROOTHUB\\_CONFIG\\_INIT](#) initialization function (see Ucxclass.h).

## Syntax

C++

```
typedef struct _UCX_ROOTHUB_CONFIG {
    ULONG                      Size;
    ULONG                      NumberOfPresentedControlUrbCallbacks;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubClearHubFeature;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubClearPortFeature;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubGetHubStatus;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubGetPortStatus;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubSetHubFeature;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubSetPortFeature;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubGetPortErrorCount;
    PFN_UCX_ROOTHUB_CONTROL_URB EvtRootHubControlUrb;
    PFN_UCX_ROOTHUB_INTERRUPT_TX EvtRootHubInterruptTx;
    PFN_UCX_ROOTHUB_GET_INFO    EvtRootHubGetInfo;
    PFN_UCX_ROOTHUB_GET_20PORT_INFO EvtRootHubGet20PortInfo;
    PFN_UCX_ROOTHUB_GET_30PORT_INFO EvtRootHubGet30PortInfo;
    WDF_OBJECT_ATTRIBUTES        WdfRequestAttributes;
} UCX_ROOTHUB_CONFIG, *PUCX_ROOTHUB_CONFIG;
```

## Members

Size

The size in bytes of this structure.

NumberOfPresentedControlUrbCallbacks

The number of control requests sent to the default endpoint.

EvtRootHubClearHubFeature

A pointer to the [EVT\\_UCX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubClearPortFeature`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubGetHubStatus`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubGetPortStatus`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubSetHubFeature`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubSetPortFeature`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubGetPortErrorCount`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubControlUrb`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_CONTROL\\_URB](#) callback function.

`EvtRootHubInterruptTx`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_INTERRUPT\\_TX](#) callback function.

`EvtRootHubGetInfo`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_GET\\_INFO](#) callback function.

`EvtRootHubGet20PortInfo`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_GET\\_20PORT\\_INFO](#) callback function.

`EvtRootHubGet30PortInfo`

A pointer to the [EVT\\_UCHX\\_ROOTHUB\\_GET\\_30PORT\\_INFO](#) callback function.

`WdfRequestAttributes`

A pointer to a [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that specifies initialization parameters.

# Requirements

 Expand table

Requirement	Value
Header	ucxroothub.h (include Ucxclass.h)

# UcxRootHubPortChanged function (ucxroothub.h)

Article02/22/2024

Notifies UCX about a new port change event on the host controller.

## Syntax

C++

```
void UcxRootHubPortChanged(
    [in] UCXROOTHUB UcxRootHub
);
```

## Parameters

[in] UcxRootHub

A handle to the root hub object. The client driver retrieved the handle in a previous call to [UcxRootHubCreate](#).

## Return value

None

## Remarks

This method causes interrupt transfers to be sent to the host controller. UCX invokes the client driver's implementation of the [EVT\\_UCX\\_ROOTHUB\\_INTERRUPT\\_TX](#) event callback.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxroothub.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxRootHubCreate](#)

# ucxsstreams.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ucxsstreams.h contains the following programming interfaces:

## Functions

### [UcxStaticStreamsCreate](#)

Creates a static streams object.

### [UcxStaticStreamsSetStreamInfo](#)

Sets stream information for each stream enabled by the client driver.

## Structures

### [STREAM\\_INFO](#)

This structure stores information about a stream associated with a bulk endpoint.

# STREAM\_INFO structure (ucxsstreams.h)

Article02/22/2024

This structure stores information about a stream associated with a bulk endpoint.

## Syntax

C++

```
typedef struct _STREAM_INFO {
    ULONG      Size;
    WDFQUEUE  WdfQueue;
    ULONG      StreamId;
} STREAM_INFO, *PSTREAM_INFO;
```

## Members

Size

The size in bytes of this structure.

WdfQueue

A handle to the framework queue object that contains streams.

StreamId

The stream identifier. The open-static streams request obtains stream identifiers that are assigned by the USB driver stack.

## Requirements

[+] Expand table

Requirement	Value
Header	ucxsstreams.h (include Ucxclass.h, Ucxstreams.h)

# UcxStaticStreamsCreate function (ucxstreams.h)

Article02/22/2024

Creates a static streams object.

## Syntax

C++

```
NTSTATUS UcxStaticStreamsCreate(
    [in]          UCXENDPOINT           Endpoint,
                  PUCXSSTREAMS_INIT     *StaticStreamsInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
                           UCXSSTREAMS           *StaticStreams
);
```

## Parameters

`[in] Endpoint`

A handle to the endpoint object that supports static streams. The client driver retrieved the handle in a previous call to [UcxEndpointCreate](#).

`StaticStreamsInit`

A pointer to a **UCXSSTREAMS\_INIT** structure that describes various configuration operations for creating the stream object. The driver specifies function pointers to its callback functions in this structure. This structure is managed by UCX.

`[in, optional] Attributes`

A pointer to a caller-allocated **WDF\_OBJECT\_ATTRIBUTES** structure that specifies attributes for the stream object.

`StaticStreams`

A pointer to a variable that receives a handle to the new stream object.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return one an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the host controller must call this method after the [WdfDeviceCreate](#) call. The parent of the new endpoint object is the endpoint object.

Typically, the client driver calls this method in its implementation of the [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#) event callback.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxsstreams.h (include Ucxclass.h, Ucxstreams.h)
IRQL	PASSIVE_LEVEL

# UcxStaticStreamsSetStreamInfo function (ucxstreams.h)

Article02/22/2024

Sets stream information for each stream enabled by the client driver.

## Syntax

C++

```
void UcxStaticStreamsSetStreamInfo(
    [in] UCXSSTREAMS StaticStreams,
    [in] PSTREAM_INFO StreamInfo
);
```

## Parameters

[in] StaticStreams

The handle to the Static Streams object just been created.

[in] StreamInfo

A pointer to a [STREAM\\_INFO](#) structure that contains static stream-related information.

## Return value

None

## Remarks

The client driver must call this method from its implementation of the [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ADD](#) event callback . This method must be called for the number of streams on the endpoint.

For a code example, see [EVT\\_UCX\\_ENDPOINT\\_STATIC\\_STREAMS\\_ADD](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxsstreams.h (include Ucxclass.h, Ucxstreams.h)

## See also

[UcxEndpointCreate](#)

# ucxusbdevice.h header

Article01/23/2023

This header is used to write a USB host controller driver. 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.

Do not include this header directly. Instead include Ucxclass.h

For more information, see:

- [Developing Windows drivers for USB host controllers](#)
- [Universal Serial Bus \(USB\)](#)

ucxusbdevice.h contains the following programming interfaces:

## Functions

<a href="#">UCX_USBDEVICE_EVENT_CALLBACKS_INIT</a>
Initializes a UCX_USBDEVICE_EVENT_CALLBACKS structure with the function pointers to client driver's callback functions.
<a href="#">UcxUsbDeviceCreate</a>
Creates a USB device object on the specified controller.
<a href="#">UcxUsbDeviceInitSetEventCallbacks</a>
Initializes a UCXUSBDEVICE_INIT structure with client driver's event callback functions.
<a href="#">UcxUsbDeviceRemoteWakeNotification</a>
Notifies UCX that a remote wake signal from the device is received.

## Callback functions

## [EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#)

The client driver's implementation that UCX calls to address the USB device.

## [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

The client driver's implementation that UCX calls to add a new default endpoint for a USB device.

## [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#)

The client driver's implementation that UCX calls to release controller resources associated with the device and its default endpoint.

## [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#)

The client driver's implementation that UCX calls to program information about the device and its default control endpoint into the controller.

## [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#)

The client driver's implementation that UCX calls to add a new endpoint for a USB device.

## [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#)

The client driver's implementation that UCX calls to configure endpoints in the controller.

## [EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#)

UCX invokes this callback to retrieve the device characteristics.

## [EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#)

The client driver's implementation that UCX calls to retrieve hub properties.

## [EVT\\_UCX\\_USBDEVICE\\_RESET](#)

The client driver's implementation that UCX calls when the port to which the device is attached is reset.

## [EVT\\_UCX\\_USBDEVICE\\_RESUME](#)

UCX invokes this callback function to resume a device from suspend state.

## [EVT\\_UCX\\_USBDEVICE\\_SUSPEND](#)

UCX invokes this callback function to send a device suspend state.

## [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#)

The client driver's implementation that UCX calls to update device properties.

## Structures

### [ADDRESS0\\_OWNERSHIP\\_ACQUIRE](#)

Contains parameters for configuring the device.

### [UCX\\_USBDEVICE\\_CHARACTERISTIC](#)

Stores the characteristics of an device.

### [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_PATH\\_DELAY](#)

Learn how UCX\_USBDEVICE\_CHARACTERISTIC\_PATH\_DELAY stores the isochronous transfer path delay values.

### [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#)

This structure provides a list of UCX USB device event callback functions.

### [UCXUSBDEVICE\\_INFO](#)

Contains information about the USB device. This structure is passed by UCX in the EVT\_UCX\_CONTROLLER\_USBDEVICE\_ADD event callback function.

### [USB\\_DEVICE\\_PORT\\_PATH](#)

Contains the port path of a USB device.

### [USBDEVICE\\_ABORTIO](#)

Contains a handle for the Universal Serial Bus (USB) hub or device for which to abort data transfers.

### [USBDEVICE\\_ADDRESS](#)

Contains parameters for a request to transition the specified device to the Addressed state. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_ADDRESS callback function.

### [USBDEVICE\\_DISABLE](#)

Contains parameters for a request to disable the specified device. This structure is passed by UCX

in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_DISABLE callback function.

#### [USBDEVICE\\_ENABLE](#)

Contains parameters for a request to enable the specified device. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_ENABLE callback function.

#### [USBDEVICE\\_ENABLE\\_FAILURE\\_FLAGS](#)

The flags that are set by the client driver in the EVT\_UCX\_USBDEVICE\_ENABLE callback function. Indicate errors, if any, that might have occurred while enabling the device.

#### [USBDEVICE\\_HUB\\_INFO](#)

Contains parameters for a request to get information about the specified hub. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_HUB\_INFO callback function.

#### [USBDEVICE\\_MGMT\\_HEADER](#)

This structure provides a handle for the Universal Serial Bus (USB) hub or device physically connected to the bus.

#### [USBDEVICE\\_PURGEIO](#)

The USBDEVICE\_PURGEIO structure contains the handle for the Universal Serial Bus (USB) hub or device to purge I/O for.

#### [USBDEVICE\\_RESET](#)

Contains parameters for a request to reset the specified device. This structure is passed by UCX in request parameters (Parameters.Others.Arg1) of a framework request object of the EVT\_UCX\_USBDEVICE\_RESET callback function.

#### [USBDEVICE\\_STARTIO](#)

Contains a handle for the Universal Serial Bus (USB) hub or device on which to start data transfer.

#### [USBDEVICE\\_TREE\\_PURGEIO](#)

This structure provides the handle for the Universal Serial Bus (USB) device tree to purge I/O for.

#### [USBDEVICE\\_UPDATE](#)

Passed by UCX to update the specified device. This structure is in the request parameters (Parameters.Others.Arg1) of a framework request object passed in the EVT\_UCX\_USBDEVICE\_UPDATE callback function.

### [USBDEVICE\\_UPDATE\\_20\\_HARDWARE\\_LPM\\_PARAMETERS](#)

Contains parameters for a request to update USB 2.0 link power management (LPM). UCX passes this structure in the EVT\_UCX\_USBDEVICE\_UPDATE callback function.

### [USBDEVICE\\_UPDATE\\_FAILURE\\_FLAGS](#)

The flags that are set by the client driver in the EVT\_UCX\_USBDEVICE\_UPDATE callback function. Indicate errors, if any, that might have occurred while updating the device.

### [USBDEVICE\\_UPDATE\\_FLAGS](#)

Contains request flags set by UCX that is passed in the USBDEVICE\_UPDATE structure when UCX invokes the client driver's EVT\_UCX\_USBDEVICE\_UPDATE callback function.

## Enumerations

### [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_TYPE](#)

Defines values that indicates the type of device characteristic.

### [UCX\\_USBDEVICE\\_RECOVERY\\_ACTION](#)

Defines values for FLDR and PLDR trigger resets.

# ADDRESS0\_OWNERSHIP\_ACQUIRE structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for configuring the device.

## Syntax

C++

```
typedef struct _ADDRESS0_OWNERSHIP_ACQUIRE {
    USBDEVICE_MGMT_HEADER Header;
} ADDRESS0_OWNERSHIP_ACQUIRE, *PADDRESS0_OWNERSHIP_ACQUIRE;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

# EVT\_UCX\_USBDEVICE\_ADDRESS callback function (ucxusbdevice.h)

Article02/22/2024

The client driver's implementation that UCX calls to address the USB device.

## Syntax

C++

```
EVT_UCX_USBDEVICE_ADDRESS EvtUcxUsbdeviceAddress;

void EvtUcxUsbdeviceAddress(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `Request`

A structure of type [USBDEVICE\\_ADDRESS](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceAddress(
    UCXCONTROLLER      UcxController,
    WDFREQUEST         Request
)
{

    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceAddress");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);

    usbDeviceAddress =
(PUSBDEVICE_ADDRESS)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#)

[UcxUsbDeviceCreate](#)

[UcxUsbDeviceInitSetEventCallbacks](#)

# EVT\_UCX\_USBDEVICE\_DEFAULT\_ENDPOINT\_ADD callback function (ucxusbdevice.h)

Article02/22/2024

The client driver's implementation that UCX calls to add a new default endpoint for a USB device.

## Syntax

C++

```
EVT_UCX_USBDEVICE_DEFAULT_ENDPOINT_ADD EvtUcxUsbdeviceDefaultEndpointAdd;

NTSTATUS EvtUcxUsbdeviceDefaultEndpointAdd(
    [in] UCXCONTROLLER UcxController,
    [in] UCXUSBDEVICE UcxUsbDevice,
    [in] ULONG MaxPacketSize,
    [in] PUCXENDPOINT_INIT UcxEndpointInit
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `UcxUsbDevice`

A handle to a UCX object that represents the USB device.

[in] `MaxPacketSize`

Maximum packet size for transfers on this endpoint.

[in] `UcxEndpointInit`

A pointer to an opaque structure containing initialization information. Callbacks for the endpoint object are associated with this structure. This structure is managed by UCX.

# Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

The callback function calls [UcxEndpointCreate](#) to create a new default endpoint object and register its default endpoint object callback functions.

Then, the callback function typically creates a WDF queue associated with the endpoint object. The queue does not receive any requests until the class extension starts it.

## Examples

C++

```
NTSTATUS  
Endpoint_EvtUcxUsbDeviceDefaultEndpointAdd(  
    UCXCONTROLLER          UcxController,  
    UCXUSBDEVICE           UcxUsbDevice,  
    ULONG                  MaxPacketSize,  
    PUCXENDPOINT_INIT     EndpointInit  
)  
  
{  
    NTSTATUS                status = STATUS_SUCCESS;  
  
    UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS  
    ucxDefaultEndpointEventCallbacks;  
    WDF_OBJECT_ATTRIBUTES      objectAttributes;  
  
    PUCX_CONTROLLER_CONTEXT   ucxControllerContext;  
  
    UCXENDPOINT              ucxEndpoint;  
    PUCX_ENDPOINT_CONTEXT     ucxEndpointContext;  
  
    WDF_IO_QUEUE_CONFIG       queueConfig;  
    WDFQUEUE                 wdfQueue;  
  
    UCX_DEFAULT_ENDPOINT_EVENT_CALLBACKS_INIT(&ucxDefaultEndpointEventCallbacks,
```

```

        Endpoint_EvtUcxEndpointPurge,
        Endpoint_EvtUcxEndpointStart,
        Endpoint_EvtUcxEndpointAbort,

    Endpoint_EvtUcxEndpointOkToCancelTransfers,

    Endpoint_EvtUcxDefaultEndpointUpdate);

    UcxDefaultEndpointInitSetEventCallbacks(EndpointInit,
&ucxDefaultEndpointEventCallbacks);

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes,
UCX_ENDPOINT_CONTEXT);

    ucxControllerContext = GetUcxControllerContext(UcxController);

    status = UcxEndpointCreate(UcxUsbDevice,
        &EndpointInit,
        &objectAttributes,
        &ucxEndpoint);

    if (!NT_SUCCESS(status)) {
        DbgTrace(TL_ERROR, Endpoint, "UcxEndpoint Failed %!STATUS!", status);
        goto EvtUsbDeviceDefaultEndpointAddEnd;
    }

    DbgTrace(TL_INFO, Endpoint, "UcxEndpoint created");

    ucxEndpointContext = GetUcxEndpointContext(ucxEndpoint);

    ucxEndpointContext->IsDefault = TRUE;
    ucxEndpointContext->MaxPacketSize = MaxPacketSize;

    WDF_IO_QUEUE_CONFIG_INIT(&queueConfig, WdfIoQueueDispatchManual);

    status = WdfIoQueueCreate(ucxControllerContext->WdfDevice,
        &queueConfig,
        WDF_NO_OBJECT_ATTRIBUTES,
        &wdfQueue);

    if (!NT_SUCCESS(status)) {
        DbgTrace(TL_ERROR, Endpoint, "WdfIoQueueCreate Failed %!STATUS!", status);
        goto EvtUsbDeviceDefaultEndpointAddEnd;
    }

    UcxEndpointSetWdfIoQueue(ucxEndpoint, wdfQueue);

EvtUsbDeviceDefaultEndpointAddEnd:

    return status;
}

```

# Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

- [UCX\\_DEFAULT\\_ENDPOINT\\_EVENT\\_CALLBACKS\\_INIT](#)
- [UcxDefaultEndpointInitSetEventCallbacks](#)
- [UcxEndpointCreate](#)
- [UcxUsbDeviceCreate](#)
- [WDF\\_IO\\_QUEUE\\_CONFIG\\_INIT](#)
- [WdfIoQueueCreate](#)

# EVT\_UCX\_USBDEVICE\_DISABLE callback function (ucxusbdevice.h)

Article 10/21/2021

The client driver's implementation that UCX calls to release controller resources associated with the device and its default endpoint.

## Syntax

C++

```
EVT_UCX_USBDEVICE_DISABLE EvtUcxUsbdeviceDisable;

void EvtUcxUsbdeviceDisable(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `Request`

A structure of type [USBDEVICE\\_DISABLE](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

When the client driver has released controller resources, it completes the WDFREQUEST. After completion, the only callback function that UCX calls referencing this USB device is [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#).

While the device is disabled, UCX does not schedule transfers for it.

To transition the device to the desired state, the host controller driver communicates with the hardware to complete the request.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceDisable(
    UCXCONTROLLER    UcxController,
    WDFREQUEST       Request
)

{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceDisable");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);
    usbDeviceDisable =
(PUSBDEVICE_DISABLE)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0

<b>Header</b>	ucxusbdevice.h (include Ucxclass.h)
<b>IRQL</b>	DISPATCH_LEVEL

## See also

[UcxUsbDeviceCreate](#)

# EVT\_UCX\_USBDEVICE\_ENABLE callback function (ucxusbdevice.h)

Article 10/21/2021

The client driver's implementation that UCX calls to program information about the device and its default control endpoint into the controller.

## Syntax

C++

```
EVT_UCX_USBDEVICE_ENABLE EvtUcxUsbdeviceEnable;

void EvtUcxUsbdeviceEnable(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `Request`

A structure of type [USBDEVICE\\_ENABLE](#).

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

To transition the device to the desired state, the host controller driver communicates with the hardware to complete the request.

In this callback function, the client driver prepares the controller to accept and schedule transfers on the default control endpoint for the USB device.

When the driver has finished, it completes the WDFREQUEST.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceEnable(
    UCXCONTROLLER      UcxController,
    WDFREQUEST         Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceEnable");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);
    usbDeviceEnable =
(PUSBDEVICE_ENABLE)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[UcxUsbDeviceCreate](#)

# EVT\_UCX\_USBDEVICE\_ENDPOINT\_ADD callback function (ucxusbdevice.h)

Article 02/22/2024

The client driver's implementation that UCX calls to add a new endpoint for a USB device.

## Syntax

C++

```
EVT_UCX_USBDEVICE_ENDPOINT_ADD EvtUcxUsbdeviceEndpointAdd;

NTSTATUS EvtUcxUsbdeviceEndpointAdd(
    [in]          UCXCONTROLLER UcxController,
    [in]          UCXUSBDEVICE UcxUsbDevice,
    [in]          PUSB_ENDPOINT_DESCRIPTOR UsbEndpointDescriptor,
    [in]          ULONG UsbEndpointDescriptorBufferLength,
    [in, optional] PUSB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR
SuperSpeedEndpointCompanionDescriptor,
    [in]          PUCXENDPOINT_INIT UcxEndpointInit
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxUsbDevice

A handle to a UCX object that represents the USB device.

[in] UsbEndpointDescriptor

A pointer to a location containing a USB descriptor for the endpoint being created.

[in] UsbEndpointDescriptorBufferLength

Length in bytes of the descriptor.

[in, optional] SuperSpeedEndpointCompanionDescriptor

An additional descriptor for a super speed port. This parameter is optional and may be **NULL**.

[in] UcxEndpointInit

A pointer to an opaque structure containing initialization information. Callbacks for the endpoint object are associated with this structure. This structure is managed by UCX.

# Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

The callback function calls [UcxEndpointCreate](#) to create a new endpoint object and register endpoint object callback functions.

Then, the callback function typically creates a WDF queue associated with the endpoint object. The queue does not receive any requests until the class extension starts it.

## Examples

```

UCX_ENDPOINT_EVENT_CALLBACKS      ucxEndpointEventCallbacks;
WDF_OBJECT_ATTRIBUTES            objectAttributes;

PUCX_CONTROLLER_CONTEXT         ucxControllerContext;

UCXENDPOINT                     ucxEndpoint;
PUCX_ENDPOINT_CONTEXT           ucxEndpointContext;

WDF_IO_QUEUE_CONFIG              queueConfig;

UNREFERENCED_PARAMETER(UsbEndpointDescriptor);
UNREFERENCED_PARAMETER(UsbEndpointDescriptorBufferLength);
UNREFERENCED_PARAMETER(SuperSpeedEndpointCompanionDescriptor);

UCX_ENDPOINT_EVENT_CALLBACKS_INIT(&ucxEndpointEventCallbacks,
                                  Endpoint_EvtUcxEndpointPurge,
                                  Endpoint_EvtUcxEndpointStart,
                                  Endpoint_EvtUcxEndpointAbort,
                                  Endpoint_EvtUcxEndpointReset,
                                  Endpoint_EvtUcxEndpointOkToCancelTransfers,
                                  Endpoint_EvtUcxEndpointStaticStreamsAdd,
                                  Endpoint_EvtUcxEndpointStaticStreamsEnable,
                                  Endpoint_EvtUcxEndpointStaticStreamsDisable,
                                  Endpoint_EvtUcxEndpointEnableForwardProgress);

UcxEndpointInitSetEventCallbacks(UcxEndpointInit,
&ucxEndpointEventCallbacks);

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes,
UCX_ENDPOINT_CONTEXT);

ucxControllerContext = GetUcxControllerContext(UcxController);

status = UcxEndpointCreate(UcxUsbDevice,
                           &UcxEndpointInit,
                           &objectAttributes,
                           &ucxEndpoint);

if (!NT_SUCCESS(status)) {
    DbgTrace(TL_ERROR, Endpoint, "UcxEndpoint Failed %!STATUS!", status);
    goto EvtUsbDeviceEndpointAddEnd;
}

DbgTrace(TL_INFO, Endpoint, "UcxEndpoint created");

ucxEndpointContext = GetUcxEndpointContext(ucxEndpoint);

ucxEndpointContext->IsDefault = FALSE;

```

```

ucxEndpointContext->MaxPacketSize = MAX_PACKET_SIZE;

WDF_IO_QUEUE_CONFIG_INIT(&queueConfig, WdfIoQueueDispatchManual);

status = WdfIoQueueCreate(ucxControllerContext->WdfDevice,
    &queueConfig,
    WDF_NO_OBJECT_ATTRIBUTES,
    &ucxEndpointContext->IoQueue);

if (!NT_SUCCESS(status)) {
    DbgTrace(TL_ERROR, Endpoint, "WdfIoQueueCreate Failed %!STATUS!", status);
    goto EvtUsbDeviceEndpointAddEnd;
}

UcxEndpointSetWdfIoQueue(ucxEndpoint, ucxEndpointContext->IoQueue);

EvtUsbDeviceEndpointAddEnd:

return status;
}

```

## Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[UcxDefaultEndpointInitSetEventCallbacks](#)

[UcxEndpointCreate](#)

[UcxUsbDeviceCreate](#)

[WDF\\_IO\\_QUEUE\\_CONFIG\\_INIT](#)

## [WdfIoQueueCreate](#)

# EVT\_UCX\_USBDEVICE\_ENDPOINTS\_CONFIGURE callback function (ucxusbdevice.h)

Article10/21/2021

The client driver's implementation that UCX calls to configure endpoints in the controller.

## Syntax

C++

```
EVT_UCX_USBDEVICE_ENDPOINTS_CONFIGURE EvtUcxUsbdeviceEndpointsConfigure;

void EvtUcxUsbdeviceEndpointsConfigure(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] Request

Contains a structure of type [ENDPOINTS\\_CONFIGURE](#) structure.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

In the callback, the driver programs or deprograms the endpoints, as described in the [ENDPOINTS\\_CONFIGURE](#) structure.

This callback does not enable or disable the default endpoint. The default endpoint's state is tied to the state of the device. The driver implements enable and disable operations in the [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#) and [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#) callback functions.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceEndpointsConfigure(
    UCXCONTROLLER      UcxController,
    WDFREQUEST         Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice,
        "UsbDevice_EvtUcxUsbDeviceEndpointsConfigure");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);

    ...

    endpointsConfigure =
(PENDPOINTS_CONFIGURE)wdfRequestParams.Parameters.Others.Arg1;

    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

Target Platform	Windows
Minimum KMDF version	1.0

Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[ENDPOINTS\\_CONFIGURE](#)

[UcxUsbDeviceCreate](#)

# EVT\_UCX\_USBDEVICE\_GET\_CHARACTERISTIC callback function (ucxusbdevice.h)

Article02/22/2024

UCX invokes this callback to retrieve the device characteristics.

## Syntax

C++

```
EVT_UCX_USBDEVICE_GET_CHARACTERISTIC EvtUcxUsbdeviceGetCharacteristic;

NTSTATUS EvtUcxUsbdeviceGetCharacteristic(
    [in]      UCXCONTROLLER UcxController,
    [in]      UCXUSBDEVICE UcxUsbDevice,
    [in, out] PUCX_USBDEVICE_CHARACTERISTIC UcxUsbDeviceCharacteristic
)
{...}
```

## Parameters

[in] `UcxController`

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] `UcxUsbDevice`

A handle to a UCX object that represents the USB device that the client driver received in a previous call to the [UcxUsbDeviceCreate](#) method.

[in, out] `UcxUsbDeviceCharacteristic`

A pointer to a [UCX\\_USBDEVICE\\_CHARACTERISTIC](#) structure that contains the type of characteristic in which the caller is interested. The client driver fills the value of the requested characteristic. For example, if the type indicates

[UCX\\_USBDEVICE\\_CHARACTERISTIC\\_PATH\\_DELAY](#), the driver fills the [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_PATH\\_DELAY](#) structure, pointed to by `PathDelay` member, with the appropriate maximum and send path delay values.

# Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The UCX client driver registers its implementation with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

# EVT\_UCX\_USBDEVICE\_HUB\_INFO callback function (ucxusbdevice.h)

Article02/22/2024

The client driver's implementation that UCX calls to retrieve hub properties.

## Syntax

C++

```
EVT_UCX_USBDEVICE_HUB_INFO EvtUcxUsbdeviceHubInfo;

void EvtUcxUsbdeviceHubInfo(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] Request

Contains the [USBDEVICE\\_HUB\\_INFO](#) structure.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

If the USB device is not a hub, do not provide this callback.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceHubInfo(
    UCXCONTROLLER        UcxController,
    WDFREQUEST           Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceHubInfo");

    //
    // Retrieve the USBDEVICE_HUB_INFO pointer from the
    // IOCTL_INTERNAL_USB_USBDEVICE_HUB_INFO WdfRequest.
    //
    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);

    hubInfo = (PUSBDEVICE_HUB_INFO)wdfRequestParams.Parameters.Others.Arg1;

    ....

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#)

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#)

[USBDEVICE\\_HUB\\_INFO](#)

[UcxUsbDeviceCreate](#)

# EVT\_UCX\_USBDEVICE\_RESET callback function (ucxusbdevice.h)

Article02/22/2024

The client driver's implementation that UCX calls when the port to which the device is attached is reset.

## Syntax

C++

```
EVT_UCX_USBDEVICE_RESET EvtUcxUsbdeviceReset;

void EvtUcxUsbdeviceReset(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] Request

Contains the [USBDEVICE\\_RESET](#) structure.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

To transition the device to the desired state, the host controller driver communicates with the hardware to complete the request.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceReset(
    UCXCONTROLLER        UcxController,
    WDFREQUEST           Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceReset");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);
    usbDeviceReset =
(PUSBDEVICE_RESET)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[USBDEVICE\\_RESET](#)

[UcxUsbDeviceCreate](#)

# EVT\_UCX\_USBDEVICE\_RESUME callback function (ucxusbdevice.h)

Article02/22/2024

UCX invokes this callback function to resume a device from suspend state.

## Syntax

C++

```
EVT_UCX_USBDEVICE_RESUME EvtUcxUsbdeviceResume;

void EvtUcxUsbdeviceResume(
    [in] UCXCONTROLLER UcxController,
    [in] UCXUSBDEVICE UcxUsbDevice
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxUsbDevice

A handle to a UCX object that represents the USB device that the client driver received in a previous call to the [UcxUsbDeviceCreate](#) method.

## Return value

None

## Remarks

The UCX client driver registers its implementation with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

# EVT\_UCX\_USBDEVICE\_SUSPEND callback function (ucxusbdevice.h)

Article 02/22/2024

UCX invokes this callback function to send a device suspend state.

## Syntax

C++

```
EVT_UCX_USBDEVICE_SUSPEND EvtUcxUsbdeviceSuspend;

void EvtUcxUsbdeviceSuspend(
    [in] UCXCONTROLLER UcxController,
    [in] UCXUSBDEVICE UcxUsbDevice
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] UcxUsbDevice

A handle to a UCX object that represents the USB device that the client driver received in a previous call to the [UcxUsbDeviceCreate](#) method.

## Return value

None

## Remarks

The UCX client driver registers its implementation with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

# EVT\_UCX\_USBDEVICE\_UPDATE callback function (ucxusbdevice.h)

Article02/22/2024

The client driver's implementation that UCX calls to update device properties.

## Syntax

C++

```
EVT_UCX_USBDEVICE_UPDATE EvtUcxUsbdeviceUpdate;

void EvtUcxUsbdeviceUpdate(
    [in] UCXCONTROLLER UcxController,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] UcxController

A handle to the UCX controller that the client driver received in a previous call to the [UcxControllerCreate](#) method.

[in] Request

Contains the [USBDEVICE\\_UPDATE](#) structure.

## Return value

None

## Remarks

The UCX client driver registers this callback function with the USB host controller extension (UCX) by calling the [UcxUsbDeviceCreate](#) method.

The host controller driver communicates with the hardware to update descriptors, LPM parameters, whether device is a hub, and maximum exit latency, as needed.

The client driver returns completion status in *Request*. The driver can complete the WDFREQUEST asynchronously.

## Examples

```
VOID
UsbDevice_EvtUcxUsbDeviceUpdate(
    UCXCONTROLLER        UcxController,
    WDFREQUEST           Request
)
{
    UNREFERENCED_PARAMETER(UcxController);

    DbgTrace(TL_INFO, UsbDevice, "UsbDevice_EvtUcxUsbDeviceUpdate");

    WDF_REQUEST_PARAMETERS_INIT(&wdfRequestParams);
    WdfRequestGetParameters(WdfRequest, &wdfRequestParams);
    usbDeviceUpdate =
(PUSBDEVICE_UPDATE)wdfRequestParams.Parameters.Others.Arg1;
    ...

    WdfRequestComplete(Request, STATUS_SUCCESS);
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	DISPATCH_LEVEL

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#)

`USBDEVICE_UPDATE`

`UcxUsbDeviceCreate`

`UcxUsbDeviceInitSetEventCallbacks`

# UCX\_USBDEVICE\_CHARACTERISTIC structure (ucxusbdevice.h)

Article02/22/2024

Stores the characteristics of an device.

## Syntax

C++

```
typedef struct _UCX_USBDEVICE_CHARACTERISTIC {
    ULONG                         Size;
    UCX_USBDEVICE_CHARACTERISTIC_TYPE CharacteristicType;
    union {
        UCX_USBDEVICE_CHARACTERISTIC_PATH_DELAY PathDelay;
    };
} UCX_USBDEVICE_CHARACTERISTIC, *PUCX_USBDEVICE_CHARACTERISTIC;
```

## Members

Size

Size of this structure.

CharacteristicType

A [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_TYPE](#)-type value that indicates the type of device characteristic.

PathDelay

A [UCX\\_USBDEVICE\\_CHARACTERISTIC\\_PATH\\_DELAY](#)-typed value that indicates the path delay values for the endpoint.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#)

# UCX\_USBDEVICE\_CHARACTERISTIC\_PATH\_DELAY structure (ucxusbdevice.h)

Article 02/22/2024

Stores the isochronous transfer path delay values.

## Syntax

C++

```
typedef struct _UCX_USBDEVICE_CHARACTERISTIC_PATH_DELAY {
    ULONG MaximumSendPathDelayInMilliSeconds;
    ULONG MaximumCompletionPathDelayInMilliSeconds;
} UCX_USBDEVICE_CHARACTERISTIC_PATH_DELAY,
*PUCX_USBDEVICE_CHARACTERISTIC_PATH_DELAY;
```

## Members

MaximumSendPathDelayInMilliSeconds

The maximum delay in milliseconds from the time the client driver's isochronous transfer is received by the USB driver stack to the time the transfer is programmed in the host controller. The host controller could either be a local host (as in case of wired USB) or it could be a remote controller as in case of Media-Agnostic USB (MA-USB). In case of MA-USB, it includes the maximum delay associated with the network medium.

MaximumCompletionPathDelayInMilliSeconds

The maximum delay in milliseconds from the time an isochronous transfer is completed by the (local or remote) host controller to the time the corresponding client driver's request is completed by the USB driver stack. For MA-USB, it includes the maximum delay associated with the network medium.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709

Requirement	Value
Minimum supported server	Windows Server 2016
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#)

# UCX\_USBDEVICE\_CHARACTERISTIC\_TYPE enumeration (ucxusbdevice.h)

Article02/22/2024

Defines values that indicates the type of device characteristic.

## Syntax

C++

```
typedef enum _UCX_USBDEVICE_CHARACTERISTIC_TYPE {
    UCX_USBDEVICE_CHARACTERISTIC_TYPE_PATH_DELAY
} UCX_USBDEVICE_CHARACTERISTIC_TYPE;
```

## Constants

[ ] Expand table

`UCX_USBDEVICE_CHARACTERISTIC_TYPE_PATH_DELAY`

The type of characteristic of the device.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#)

## **UCX\_USBDEVICE\_CHARACTERISTIC**

# UCX\_USBDEVICE\_EVENT\_CALLBACKS structure (ucxusbdevice.h)

Article04/01/2021

This structure provides a list of UCX USB device event callback functions.

## Syntax

C++

```
typedef struct _UCX_USBDEVICE_EVENT_CALLBACKS {
    ULONG Size;
    PFN_UCX_USBDEVICE_ENDPOINTS_CONFIGURE EvtUsbDeviceEndpointsConfigure;
    PFN_UCX_USBDEVICE_ENABLE EvtUsbDeviceEnable;
    PFN_UCX_USBDEVICE_DISABLE EvtUsbDeviceDisable;
    PFN_UCX_USBDEVICE_RESET EvtUsbDeviceReset;
    PFN_UCX_USBDEVICE_ADDRESS EvtUsbDeviceAddress;
    PFN_UCX_USBDEVICE_UPDATE EvtUsbDeviceUpdate;
    PFN_UCX_USBDEVICE_HUB_INFO EvtUsbDeviceHubInfo;
    PFN_UCX_USBDEVICE_DEFAULT_ENDPOINT_ADD EvtUsbDeviceDefaultEndpointAdd;
    PFN_UCX_USBDEVICE_ENDPOINT_ADD EvtUsbDeviceEndpointAdd;
    PFN_UCX_USBDEVICE_SUSPEND EvtUsbDeviceSuspend;
    PFN_UCX_USBDEVICE_RESUME EvtUsbDeviceResume;
    PFN_UCX_USBDEVICE_GET_CHARACTERISTIC EvtUsbDeviceGetCharacteristic;
} UCX_USBDEVICE_EVENT_CALLBACKS, *PUCX_USBDEVICE_EVENT_CALLBACKS;
```

## Members

### Size

The size in bytes of this structure.

### EvtUsbDeviceEndpointsConfigure

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback function.

### EvtUsbDeviceEnable

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#) callback function.

### EvtUsbDeviceDisable

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#) callback function.

`EvtUsbDeviceReset`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_RESET](#) callback function.

`EvtUsbDeviceAddress`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#) callback function.

`EvtUsbDeviceUpdate`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) callback function.

`EvtUsbDeviceHubInfo`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#) callback function.

`EvtUsbDeviceDefaultEndpointAdd`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) callback function.

`EvtUsbDeviceEndpointAdd`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#) callback function.

`EvtUsbDeviceSuspend`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_SUSPEND](#) callback function.

`EvtUsbDeviceResume`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_RESUME](#) callback function.

`EvtUsbDeviceGetCharacteristic`

A pointer to an [EVT\\_UCX\\_USBDEVICE\\_GET\\_CHARACTERISTIC](#) callback function.

## Requirements

Header	ucxusbdevice.h (include Ucxclass.h)
--------	-------------------------------------

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#)

## UcxUsbDeviceInitSetEventCallbacks

# UCX\_USBDEVICE\_EVENT\_CALLBACKS\_INIT function (ucxusbdevice.h)

Article 10/21/2021

Initializes a [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#) structure with the function pointers to client driver's callback functions.

## Syntax

C++

```
void UCX_USBDEVICE_EVENT_CALLBACKS_INIT(
    [out] PUCX_USBDEVICE_EVENT_CALLBACKS           Callbacks,
    [in]  PFN_UCX_USBDEVICE_ENDPOINTS_CONFIGURE   EvtUsbDeviceEndpointsConfigure,
    [in]  PFN_UCX_USBDEVICE_ENABLE                 EvtUsbDeviceEnable,
    [in]  PFN_UCX_USBDEVICE_DISABLE                EvtUsbDeviceDisable,
    [in]  PFN_UCX_USBDEVICE_RESET                 EvtUsbDeviceReset,
    [in]  PFN_UCX_USBDEVICE_ADDRESS               EvtUsbDeviceAddress,
    [in]  PFN_UCX_USBDEVICE_UPDATE                EvtUsbDeviceUpdate,
    [in]  PFN_UCX_USBDEVICE_HUB_INFO              EvtUsbDeviceHubInfo,
    [in]  PFN_UCX_USBDEVICE_DEFAULT_ENDPOINT_ADD EvtUsbDeviceDefaultEndpointAdd,
    [in]  PFN_UCX_USBDEVICE_ENDPOINT_ADD          EvtUsbDeviceEndpointAdd
);
```

## Parameters

[out] `Callbacks`

A pointer to a [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#) structure to initialize.

[in] `EvtUsbDeviceEndpointsConfigure`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_ENDPOINTS\\_CONFIGURE](#) event callback function.

[in] `EvtUsbDeviceEnable`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#) event callback function.

[in] `EvtUsbDeviceDisable`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#) event callback function.

[in] `EvtUsbDeviceReset`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_RESET](#) event callback function.

[in] `EvtUsbDeviceAddress`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#) event callback function.

[in] `EvtUsbDeviceUpdate`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) event callback function.

[in] `EvtUsbDeviceHubInfo`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#) event callback function.

[in] `EvtUsbDeviceDefaultEndpointAdd`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) event callback function.

[in] `EvtUsbDeviceEndpointAdd`

A pointer to client driver's implementation of the [EVT\\_UCX\\_USBDEVICE\\_ENDPOINT\\_ADD](#) event callback function.

## Return value

None

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows

Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#)

[UcxUsbDeviceCreate](#)

[UcxUsbDeviceInitSetEventCallbacks](#)

# UCX\_USBDEVICE\_RECOVERY\_ACTION enumeration (ucxusbdevice.h)

Article02/22/2024

Defines values for FLDR and PLDR trigger resets.

## Syntax

C++

```
typedef enum _UCX_USBDEVICE_RECOVERY_ACTION {
    UcxUsbDeviceRecoverActionNone,
    UcxUsbDeviceRecoverActionFunctionLevelDeviceReset,
    UcxUsbDeviceRecoverActionPlatformLevelDeviceReset
} UCX_USBDEVICE_RECOVERY_ACTION;
```

## Constants

[ ] Expand table

<code>UcxUsbDeviceRecoverActionNone</code>	None.
<code>UcxUsbDeviceRecoverActionFunctionLevelDeviceReset</code>	FLDR trigger reset.
<code>UcxUsbDeviceRecoverActionPlatformLevelDeviceReset</code>	PLDR trigger reset.

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Header	ucxusbdevice.h (include Ucxclass.h)

# UCXUSBDEVICE\_INFO structure (ucxusbdevice.h)

Article02/22/2024

Contains information about the USB device. This structure is passed by UCX in the [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#) event callback function.

## Syntax

C++

```
typedef struct _UCXUSBDEVICE_INFO {
    ULONG             Size;
    USB_DEVICE_SPEED DeviceSpeed;
    UCXUSBDEVICE     TtHub;
    USB_DEVICE_PORT_PATH PortPath;
} UCXUSBDEVICE_INFO, *PUCXUSBDEVICE_INFO;
```

## Members

Size

The size in bytes of this structure.

DeviceSpeed

Defines the device speed of the USB device or hub.

TtHub

A handle to the USB device object that represents the TT hub.

PortPath

The port path for the USB device or hub.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxusbdevice.h)

## See also

[EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#)

# UcxUsbDeviceCreate function (ucxusbdevice.h)

Article02/22/2024

Creates a USB device object on the specified controller.

## Syntax

C++

```
NTSTATUS UcxUsbDeviceCreate(
    [in]          UCXCONTROLLER      Controller,
    [out]         PUCXUSBDEVICE_INIT *UsbDeviceInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]          UCXUSBDEVICE      *UsbDevice
);
```

## Parameters

[in] Controller

A handle to the controller object. The client driver retrieved the handle in a previous call to [UcxControllerCreate](#).

[out] UsbDeviceInit

A pointer to a **UCXUSBDEVICE\_INIT** structure that describes various configuration operations for creating the USB device object. The driver specifies function pointers to its callback functions in this structure. This structure is managed by UCX.

[in, optional] Attributes

A pointer to a caller-allocated **WDF\_OBJECT\_ATTRIBUTES** structure that specifies attributes for the USB device object.

[out] UsbDevice

A pointer to a variable that receives a handle to the new USB device object.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return one an appropriate NTSTATUS error code.

## Remarks

The client driver for the host controller must call this method after the [WdfDeviceCreate](#) call. The parent of the new USB device object is the parent hub device specified by UCX.

For a code example, see [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	PASSIVE_LEVEL

## See also

[EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#)

# UcxUsbDeviceInitSetEventCallbacks function (ucxusbdevice.h)

Article 02/22/2024

Initializes a **UCXUSBDEVICE\_INIT** structure with client driver's event callback functions.

## Syntax

C++

```
void UcxUsbDeviceInitSetEventCallbacks(
    [in, out] PUCXUSBDEVICE_INIT           UsbDeviceInit,
    [in]      PUCX_USBDEVICE_EVENT_CALLBACKS EventCallbacks
);
```

## Parameters

[in, out] UsbDeviceInit

A pointer to a **UCXUSBDEVICE\_INIT** structure that UCX passes when it invokes client driver's [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#) event callback function.

[in] EventCallbacks

A pointer to a **UCX\_USBDEVICE\_EVENT\_CALLBACKS** structure that contains function pointer to client driver's event callback functions. The client driver initializes the structure by calling [UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS\\_INIT](#).

## Return value

None

## Remarks

An initialized **UCXUSBDEVICE\_INIT** structure is used by the [UcxUsbDeviceCreate](#) method to create a USB device and register the client driver's event callback functions.

For a code example, see [EVT\\_UCX\\_CONTROLLER\\_USBDEVICE\\_ADD](#).

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[UCX\\_USBDEVICE\\_EVENT\\_CALLBACKS](#)

[UcxUsbDeviceCreate](#)

# UcxUsbDeviceRemoteWakeNotification function (ucxusbdevice.h)

Article 02/22/2024

Notifies UCX that a remote wake signal from the device is received.

## Syntax

C++

```
void UcxUsbDeviceRemoteWakeNotification(
    [in] UCXUSBDEVICE UsbDevice,
    [in] ULONG        Interface
);
```

## Parameters

[in] UsbDevice

A handle to the USB device object for which the remote wake is received. The client driver retrieved the handle in a previous call to [UcxUsbDeviceCreate](#).

[in] Interface

The interface number that sent the remote wake notification.

## Return value

None

## Remarks

This function completes the pending remote wake request from the request driver such as the hub driver or usbccgp driver. If no such request is found, this notification is ignored.

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ucxusbdevice.h (include Ucxclass.h)
IRQL	<=DISPATCH_LEVEL

## See also

[UcxUsbDeviceCreate](#)

# USB\_DEVICE\_PORT\_PATH structure (ucxusbdevice.h)

Article02/22/2024

Contains the port path of a USB device.

## Syntax

C++

```
typedef struct _USB_DEVICE_PORT_PATH {
    ULONG Size;
    ULONG PortPathDepth;
    ULONG TTHubDepth;
    ULONG PortPath[MAX_USB_DEVICE_DEPTH];
} USB_DEVICE_PORT_PATH, *PUSB_DEVICE_PORT_PATH;
```

## Members

Size

The size in bytes of this structure.

PortPathDepth

The depth of path in the USB topology tree, consisting of host controller, hubs, and devices.

TTHubDepth

The depth of path in the USB topology tree from a TT hub.

PortPath[MAX\_USB\_DEVICE\_DEPTH]

The index of connected USB port on the hub.

## Requirements

[+] Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[UCXUSBDEVICE\\_INFO](#)

# USBDEVICE\_ABORTIO structure (ucxusbdevice.h)

Article02/22/2024

Contains a handle for the Universal Serial Bus (USB) hub or device for which to abort data transfers.

## Syntax

C++

```
typedef struct _USBDEVICE_ABORTIO {
    USBDEVICE_MGMT_HEADER Header;
} USBDEVICE_ABORTIO, *PUSBDEVICE_ABORTIO;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

## Requirements

 Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_PURGEIO](#)

[USBDEVICE\\_STARTIO](#)

[WdfRequestGetParameters](#)

# USBDEVICE\_ADDRESS structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for a request to transition the specified device to the Addressed state. This structure is passed by UCX in request parameters (**Parameters.Others.Arg1**) of a framework request object of the [EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_ADDRESS {
    USBDEVICE_MGMT_HEADER Header;
    ULONG                 Reserved;
    ULONG                 Address;
} USBDEVICE_ADDRESS, *PUSBDEVICE_ADDRESS;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

Reserved

Do not use.

Address

The address of the specified the USB hub or device.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_ADDRESS](#)

# USBDEVICE\_DISABLE structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for a request to disable the specified device. This structure is passed by UCX in request parameters ([Parameters.Others.Arg1](#)) of a framework request object of the [EVT\\_UCX\\_USBDEVICE\\_DISABLE](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_DISABLE {
    USBDEVICE_MGMT_HEADER      Header;
    UCXENDPOINT                DefaultEndpoint;
    UCX_USBDEVICE_RECOVERY_ACTION UsbDeviceRecoveryAction;
} USBDEVICE_DISABLE, *PUSBDEVICE_DISABLE;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

DefaultEndpoint

A handle to the default endpoint of the USB device or hub to disable.

UsbDeviceRecoveryAction

A [UCX\\_USBDEVICE\\_RECOVERY\\_ACTION](#)-value that indicates FLDR or PLDR trigger resets.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_DISABLE](#)

[USBDEVICE\\_ENABLE](#)

[WdfRequestGetParameters](#)

# USBDEVICE\_ENABLE structure (ucxusbdevice.h)

Article 02/22/2024

Contains parameters for a request to enable the specified device. This structure is passed by UCX in request parameters ([Parameters.Others.Arg1](#)) of a framework request object of the [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_ENABLE {
    USBDEVICE_MGMT_HEADER          Header;
    UCXENDPOINT                    DefaultEndpoint;
    USBDEVICE_ENABLE_FAILURE_FLAGS FailureFlags;
} USBDEVICE_ENABLE, *PUSBDEVICE_ENABLE;
```

## Members

[Header](#)

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

[DefaultEndpoint](#)

The default endpoint for the USB hub or device to enable transfers for.

[FailureFlags](#)

The errors, if any, that occurred when attempting to enable the hub or device for transfers.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_DISABLE](#)

[WdfRequestGetParameters](#)

# USBDEVICE\_ENABLE\_FAILURE\_FLAGS structure (ucxusbdevice.h)

Article02/22/2024

The flags that are set by the client driver in the [EVT\\_UCX\\_USBDEVICE\\_ENABLE](#) callback function. Indicate errors, if any, that might have occurred while enabling the device.

## Syntax

C++

```
typedef struct _USBDEVICE_ENABLE_FAILURE_FLAGS {
    ULONG InsufficientHardwareResourcesForDefaultEndpoint : 1;
    ULONG InsufficientHardwareResourcesForDevice : 1;
    ULONG Reserved : 30;
} USBDEVICE_ENABLE_FAILURE_FLAGS;
```

## Members

`InsufficientHardwareResourcesForDefaultEndpoint`

Insufficient hardware resources for transfers to the default endpoint.

`InsufficientHardwareResourcesForDevice`

Insufficient hardware resources to enable transfers.

`Reserved`

Do not use.

## Requirements

[Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_ENABLE](#)

[USBDEVICE\\_ENABLE](#)

# USBDEVICE\_HUB\_INFO structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for a request to get information about the specified hub. This structure is passed by UCX in request parameters (**Parameters.Others.Arg1**) of a framework request object of the [EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_HUB_INFO {
    USBDEVICE_MGMT_HEADER Header;
    ULONG                 NumberOfPorts;
    ULONG                 NumberOfTTs;
    ULONG                 TTThinkTime;
} USBDEVICE_HUB_INFO, *PUSBDEVICE_HUB_INFO;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

NumberOfPorts

The count of ports available for the USB hub, filled by the client driver.

NumberOfTTs

The count of TT hubs, filled by the client driver.

TTThinkTime

The ThinkTime property of the TT hub, filled by the client driver.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_HUB\\_INFO](#)

# USBDEVICE\_MGMT\_HEADER structure (ucxusbdevice.h)

Article02/22/2024

This structure provides a handle for the Universal Serial Bus (USB) hub or device physically connected to the bus.

## Syntax

C++

```
typedef struct _USBDEVICE_MGMT_HEADER {
    ULONG      Size;
    UCXUSBDEVICE Hub;
    UCXUSBDEVICE UsbDevice;
} USBDEVICE_MGMT_HEADER, *PUSBDEVICE_MGMT_HEADER;
```

## Members

Size

The size in bytes of this structure.

Hub

The handle to the USB hub that is physically connected to the bus.

UsbDevice

The handle for the USB device that is physically connected to the bus.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

# USBDEVICE\_PURGEIO structure (ucxusbdevice.h)

Article02/22/2024

The **USBDEVICE\_PURGEIO** structure contains the handle for the Universal Serial Bus (USB) hub or device to purge I/O for.

## Syntax

C++

```
typedef struct _USBDEVICE_PURGEIO {
    USBDEVICE_MGMT_HEADER Header;
    BOOLEAN             OnSuspend;
} USBDEVICE_PURGEIO, *PUSBDEVICE_PURGEIO;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

OnSuspend

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_ABORTIO](#)

[USBDEVICE\\_STARTIO](#)

USBDEVICE\_TREE\_PURGEIO

# USBDEVICE\_RESET structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for a request to reset the specified device. This structure is passed by UCX in request parameters (**Parameters.Others.Arg1**) of a framework request object of the [EVT\\_UCX\\_USBDEVICE\\_RESET](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_RESET {
    USBDEVICE_MGMT_HEADER Header;
    UCXENDPOINT          DefaultEndpoint;
    ULONG                EndpointsToDisableCount;
    UCXENDPOINT          *EndpointsToDisable;
} USBDEVICE_RESET, *PUSBDEVICE_RESET;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

DefaultEndpoint

The default endpoint for the USB hub or device.

EndpointsToDisableCount

The number of endpoints to disable.

EndpointsToDisable

A pointer to an array of handles to endpoints to disable.

## Requirements

[ ] Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[EVT\\_UCX\\_USBDEVICE\\_RESET](#)

# USBDEVICE\_STARTIO structure (ucxusbdevice.h)

Article02/22/2024

Contains a handle for the Universal Serial Bus (USB) hub or device on which to start data transfer.

## Syntax

C++

```
typedef struct _USBDEVICE_STARTIO {
    USBDEVICE_MGMT_HEADER Header;
} USBDEVICE_STARTIO, *PUSBDEVICE_STARTIO;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

## Requirements

 Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_ABORTIO](#)

[USBDEVICE\\_PURGEIO](#)

# USBDEVICE\_TREE\_PURGEIO structure (ucxusbdevice.h)

Article 02/22/2024

This structure provides the handle for the Universal Serial Bus (USB) device tree to purge I/O for.

## Syntax

C++

```
typedef struct _USBDEVICE_TREE_PURGEIO {
    USBDEVICE_MGMT_HEADER Header;
} USBDEVICE_TREE_PURGEIO, *PUSBDEVICE_TREE_PURGEIO;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

## Requirements

 Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_PURGEIO](#)

# USBDEVICE\_UPDATE structure (ucxusbdevice.h)

Article05/22/2024

Passed by UCX to update the specified device. This structure is in the request parameters (**Parameters.Others.Arg1**) of a framework request object passed in the [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_UPDATE {
    USBDEVICE_MGMT_HEADER           Header;
    USBDEVICE_UPDATE_FLAGS          Flags;
    PUSB_DEVICE_DESCRIPTOR          DeviceDescriptor;
    PUSB_BOS_DESCRIPTOR             BosDescriptor;
    ULONG                           MaxExitLatency;
    BOOLEAN                         IsHub;
    USBDEVICE_UPDATE_FAILURE_FLAGS  FailureFlags;
    USBDEVICE_UPDATE_20_HARDWARE_LPM_PARAMETERS  Usb20HardwareLpmParameters;
    USHORT                          RootPortResumeTime;
    BOOLEAN                         IsNative;
} USBDEVICE_UPDATE, *PUSBDEVICE_UPDATE;
```

## Members

Header

A [USBDEVICE\\_MGMT\\_HEADER](#) structure that contains the handle for the USB hub or device.

Flags

A bitwise-OR of [USBDEVICE\\_UPDATE\\_FLAGS](#) values that indicates the attributes that must be updated by the client driver.

DeviceDescriptor

A pointer a [USB\\_DEVICE\\_DESCRIPTOR](#) structure that contains the device descriptor.

BosDescriptor

A pointer a **USB\_BOS\_DESCRIPTOR** structure that contains the device descriptor. See [Usbspec.h](#).

#### MaxExitLatency

The maximum exit latency period.

#### IsHub

Indicates if the USB device to update is a USB hub (TRUE) or not (FALSE).

#### FailureFlags

A **USBDEVICE\_UPDATE\_FAILURE\_FLAGS** structure that indicates the errors, if any, that occurred during the update operation.

#### Usb20HardwareLpmParameters

A **USBDEVICE\_UPDATE\_20\_HARDWARE\_LPM\_PARAMETERS** structure that describes the Link Power Management (LPM) features.

#### RootPortResumeTime

The resume time for the root port.

#### IsNative

Indicates if the USB device to update is native (TRUE) or not (FALSE).

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

- [USBDEVICE\\_UPDATE\\_20\\_HARDWARE\\_LPM\\_PARAMETERS](#)
- [USBDEVICE\\_UPDATE\\_FAILURE\\_FLAGS](#)
- [USBDEVICE\\_UPDATE\\_FLAGS](#)
- [WdfRequestGetParameters](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USBDEVICE\_UPDATE\_20\_HARDWARE\_LP M\_PARAMETERS structure (ucxusbdevice.h)

Article02/22/2024

Contains parameters for a request to update USB 2.0 link power management (LPM). UCX passes this structure in the [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_UPDATE_20_HARDWARE_LPM_PARAMETERS {  
    ULONG HardwareLpmEnable : 1;  
    ULONG RemoteWakeEnable : 1;  
    ULONG HostInitiatedResumeDurationMode : 1;  
    ULONG BestEffortServiceLatency : 4;  
    ULONG BestEffortServiceLatencyDeep : 4;  
    ULONG L1Timeout : 8;  
    ULONG Reserved : 13;  
} USBDEVICE_UPDATE_20_HARDWARE_LPM_PARAMETERS;
```

## Members

`HardwareLpmEnable`

If set, indicates are request to enable hardware LPM.

`RemoteWakeEnable`

If set, indicates are request to enable remote wake signal.

`HostInitiatedResumeDurationMode`

The requested resume period.

`BestEffortServiceLatency`

The requested best effort service latency.

`BestEffortServiceLatencyDeep`

The requested best effort service latency deep.

L1Timeout

The requested L1 timeout.

Reserved

Do not use.

## Requirements

 Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[ROOTHUB\\_20PORT\\_INFO](#)

[USBDEVICE\\_UPDATE](#)

[USBDEVICE\\_UPDATE\\_FAILURE\\_FLAGS](#)

[USBDEVICE\\_UPDATE\\_FLAGS](#)

# USBDEVICE\_UPDATE\_FAILURE\_FLAGS structure (ucxusbdevice.h)

Article02/22/2024

The flags that are set by the client driver in the [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) callback function. Indicate errors, if any, that might have occurred while updating the device.

## Syntax

C++

```
typedef struct _USBDEVICE_UPDATE_FAILURE_FLAGS {
    ULONG MaxExitLatencyTooLarge : 1;
    ULONG Reserved : 31;
} USBDEVICE_UPDATE_FAILURE_FLAGS;
```

## Members

`MaxExitLatencyTooLarge`

The maximum exit latency is larger than expected.

`Reserved`

Do not use.

## Requirements

  [Expand table](#)

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

[USBDEVICE\\_UPDATE](#)

[USBDEVICE\\_UPDATE\\_20\\_HARDWARE\\_LPM\\_PARAMETERS](#)

USBDEVICE\_UPDATE\_FLAGS

# USBDEVICE\_UPDATE\_FLAGS structure (ucxusbdevice.h)

Article05/22/2024

Contains request flags set by UCX that is passed in the [USBDEVICE\\_UPDATE](#) structure when UCX invokes the client driver's [EVT\\_UCX\\_USBDEVICE\\_UPDATE](#) callback function.

## Syntax

C++

```
typedef struct _USBDEVICE_UPDATE_FLAGS {
    ULONG UpdateDeviceDescriptor : 1;
    ULONG UpdateBosDescriptor : 1;
    ULONG UpdateMaxExitLatency : 1;
    ULONG UpdateIsHub : 1;
    ULONG UpdateAllowIoOnInvalidPipeHandles : 1;
    ULONG Update20HardwareLpmParameters : 1;
    ULONG UpdateRootPortResumeTime : 1;
    ULONG UpdateTunnelState : 1;
    ULONG Reserved : 25;
} USBDEVICE_UPDATE_FLAGS;
```

## Members

`UpdateDeviceDescriptor`

If set, indicates a request to update the USB device descriptor.

`UpdateBosDescriptor`

If set, indicates a request to update the USB BOS descriptor.

`UpdateMaxExitLatency`

If set, indicates a request to update the maximum exit latency.

`UpdateIsHub`

If set, indicates a request to determine if the device is a hub.

`UpdateAllowIoOnInvalidPipeHandles`

If set, indicates the USB device or hub has been updated to allow I/O with invalid pipe handles.

#### `Update20HardwareLpmParameters`

If set, indicates a request to update the 2.0 LPM state.

#### `UpdateRootPortResumeTime`

If set, indicates a request to update the root port resume time.

#### `UpdateTunnelState`

If set, indicates a request to update the USB tunnel state.

#### `Reserved`

Do not use.

## Requirements

[+] Expand table

Requirement	Value
Header	ucxusbdevice.h (include Ucxclass.h)

## See also

- [USBDEVICE\\_UPDATE](#)
- [USBDEVICE\\_UPDATE\\_20\\_HARDWARE\\_LPM\\_PARAMETERS](#)
- [USBDEVICE\\_UPDATE\\_FAILURE\\_FLAGS](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# udecxurb.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

udecxurb.h contains the following programming interfaces:

## Functions

<a href="#">UdecxUrbComplete</a>
Completes the URB request with a USB-specific completion status code.
<a href="#">UdecxUrbCompleteWithNtStatus</a>
Completes the URB request with an NTSTATUS code.
<a href="#">UdecxUrbRetrieveBuffer</a>
Retrieves the transfer buffer of an URB from the specified framework request object sent to the endpoint queue.
<a href="#">UdecxUrbRetrieveControlSetupPacket</a>
Retrieves a USB control setup packet from a specified framework request object.
<a href="#">UdecxUrbSetBytesCompleted</a>
Sets the number of bytes transferred for the URB contained within a framework request object.

# UdecxUrbComplete function (udecxurb.h)

Article 02/22/2024

Completes the URB request with a USB-specific completion status code.

## Syntax

C++

```
void UdecxUrbComplete(
    [in] WDFREQUEST Request,
    [in] USBD_STATUS UsbdStatus
);
```

## Parameters

[in] Request

A handle to a framework request object that contains the [URB](#) for the transfer.

[in] UsbdStatus

A [USBD\\_STATUS](#)-typed value that indicates the success or failure of the completed URB request.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxurb.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUrbCompleteWithNtStatus function (udecxurb.h)

Article 10/21/2021

Completes the URB request with an [NTSTATUS](#) code.

## Syntax

C++

```
void UdecxUrbCompleteWithNtStatus(
    [in] WDFREQUEST Request,
    [in] NTSTATUS    NtStatus
);
```

## Parameters

[in] Request

A handle to a framework request object that contains the [URB](#) for the transfer.

[in] NtStatus

A [NTSTATUS](#)-typed value that indicates the success or failure of the completed URB request.

## Return value

None

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Minimum KMDF version	1.15
Header	udecxurb.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUrbRetrieveBuffer function (udecxurb.h)

Article 02/22/2024

Retrieves the transfer buffer of an URB from the specified framework request object sent to the endpoint queue.

## Syntax

C++

```
NTSTATUS UdecxUrbRetrieveBuffer(
    [in] WDFREQUEST Request,
    [out] PUCHAR     *TransferBuffer,
    [out] PULONG     Length
);
```

## Parameters

[in] Request

A handle to a framework request object that contains the [URB](#) for the transfer.

[out] TransferBuffer

A pointer to a buffer that receives the transfer buffer of an [URB](#).

[out] Length

A ULONG variable that receives the length of the buffer pointer to by *TransferBuffer*.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

[ ] Expand table

Return code	Description
STATUS_INVALID_PARAMETER	The URB does not contain a transfer buffer.

**STATUS\_INSUFFICIENT\_RESOURCES** The transfer buffer MDL was not valid.

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxurb.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUrbSetBytesCompleted](#)

[Write a UDE client driver](#)

# UdecxUrbRetrieveControlSetupPacket function (udecxurb.h)

Article 02/22/2024

Retrieves a USB control setup packet from a specified framework request object.

## Syntax

C++

```
NTSTATUS UdecxUrbRetrieveControlSetupPacket(
    [in] WDFREQUEST Request,
    [out] PWDF_USB_CONTROL_SETUP_PACKET SetupPacket
);
```

## Parameters

[in] Request

A handle to a framework request object that represents the request containing the setup packet.

[out] SetupPacket

A [WDF\\_USB\\_CONTROL\\_SETUP\\_PACKET](#) structure that receives a setup packet describing the USB control transfer.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver can inspect contents of the setup packet to determine the standard control request that is sent to the device.

To complete the request, the driver must call [UdecxUrbCompleteWithNtStatus](#).

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxurb.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUrbSetBytesCompleted function (udecxurb.h)

Article 10/21/2021

Sets the number of bytes transferred for the URB contained within a framework request object.

## Syntax

C++

```
void UdecxUrbSetBytesCompleted(
    [in] WDFREQUEST Request,
        ULONG     BytesCompleted
);
```

## Parameters

[in] Request

A handle to a framework request object that contains the [URB](#) for the transfer.

BytesCompleted

The number of transferred bytes to set in the [URB](#). This value must not be greater than the transfer buffer length.

## Return value

None

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Minimum KMDF version	1.15
Header	udecxurb.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# udecxusbdevice.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

udecxusbdevice.h contains the following programming interfaces:

## Functions

### [UDECX\\_USB\\_DEVICE\\_CALLBACKS\\_INIT](#)

Initializes a UDECX\_USB\_DEVICE\_STATE\_CHANGE\_CALLBACKS structure before a UdecxUsbDeviceCreate call.

### [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS\\_INIT](#)

Initializes a UDECX\_USB\_DEVICE\_PLUG\_IN\_OPTIONS structure.

### [UdecxUsbDeviceCreate](#)

Creates a USB Device Emulation (UDE) device object.

### [UdecxUsbDeviceInitAddDescriptor](#)

Adds a USB descriptor to the initialization parameters used to create a virtual USB device.

### [UdecxUsbDeviceInitAddDescriptorWithIndex](#)

Learn how the UdecxUsbDeviceInitAddDescriptorWithIndex function adds a USB descriptor to the initialization parameters used to create a virtual USB device.

### [UdecxUsbDeviceInitAddStringDescriptor](#)

Adds a USB string descriptor to the initialization parameters used to create a virtual USB device.

### [UdecxUsbDeviceInitAddStringDescriptorRaw](#)

Learn how this method adds a USB string descriptor to the initialization parameters used to create a virtual USB device.

### [UdecxUsbDeviceInitAllocate](#)

Allocates memory for a UDECXUSBDEVICE\_INIT structure that is used to initialize a virtual USB device.

[UdecxUsbDeviceInitFree](#)

Releases the resources that were allocated by the UdecxUsbDeviceInitAllocate call.

[UdecxUsbDeviceInitSetEndpointsType](#)

Indicates the type of endpoint (simple or dynamic) in the initialization parameters that the client driver uses to create the virtual USB device.

[UdecxUsbDeviceInitSetSpeed](#)

Sets the USB speed of the virtual USB device to create.

[UdecxUsbDeviceInitSetStateChangeCallbacks](#)

Initializes a WDF-allocated structure with pointers to callback functions.

[UdecxUsbDeviceLinkPowerEntryComplete](#)

Completes an asynchronous request for bringing the device out of a low power state.

[UdecxUsbDeviceLinkPowerExitComplete](#)

Completes an asynchronous request for sending the device to a low power state.

[UdecxUsbDevicePlugIn](#)

Notifies the USB device emulation class extension (UdeCx) that the USB device has been plugged in the specified port.

[UdecxUsbDevicePlugOutAndDelete](#)

Disconnects the virtual USB device.

[UdecxUsbDeviceSetFunctionSuspendAndWakeComplete](#)

Completes an asynchronous request for changing the power state of a particular function of a virtual USB 3.0 device.

[UdecxUsbDeviceSignalFunctionWake](#)

Initiates wake up of the specified function from a low power state. This applies to virtual USB 3.0 devices.

[UdecxUsbDeviceSignalWake](#)

Initiates wake up from a low link power state for a virtual USB 2.0 device.

## Callback functions

### [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to bring the virtual USB device out of a low power state to working state.

### [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to send the virtual USB device to a low power state.

### [EVT\\_UDECX\\_USB\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create the default control endpoint on the virtual USB device.

### [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINT\\_ADD](#)

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create a dynamic endpoint on the virtual USB device.

### [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)

The USB device emulation class extension (UdeCx) invokes this callback function to change the configuration by selecting an alternate setting, disabling current endpoints, or adding dynamic endpoints.

### [EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#)

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to change the function state of the specified interface of the virtual USB 3.0 device.

## Structures

### [UDECX\\_ENDPOINTS\\_CONFIGURE\\_PARAMS](#)

Contains the configuration options specified by USB device emulation class extension (UdeCx) to

the client driver when the class extension invokes EVT\_UDECX\_USB\_DEVICE\_ENDPOINTS\_CONFIGURE.

#### [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS](#)

Contains the port numbers to which a virtual USB device is connected. Initialize this structure by calling the UDECX\_USB\_DEVICE\_PLUG\_IN\_OPTIONS\_INIT method.

#### [UDECX\\_USB\\_DEVICE\\_STATE\\_CHANGE\\_CALLBACKS](#)

Initializes a UDECX\_USB\_DEVICE\_STATE\_CHANGE\_CALLBACKS structure with pointers to callback functions that are implemented by a UDE client for a virtual USB device.

#### [UDECX\\_USB\\_ENDPOINT\\_INIT\\_AND\\_METADATA](#)

Contains the descriptors supported by an endpoint of a virtual USB device.

## Enumerations

#### [UDECX\\_ENDPOINT\\_TYPE](#)

Defines values for endpoint types supported by a virtual USB device.

#### [UDECX\\_ENDPOINTS\\_CONFIGURE\\_TYPE](#)

Defines values for endpoint configuration options.

#### [UDECX\\_USB\\_DEVICE\\_FUNCTION\\_POWER](#)

Defines values for function wake capability of a virtual USB 3.0 device.

#### [UDECX\\_USB\\_DEVICE\\_SPEED](#)

Defines values for USB device speeds.

#### [UDECX\\_USB\\_DEVICE\\_WAKE\\_SETTING](#)

Defines values for remote wake capability of a virtual USB device.

# EVT\_UDECX\_USB\_DEVICE\_D0\_ENTRY callback function (udecxusbdevice.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to bring the virtual USB device out of a low power state to working state.

## Syntax

C++

```
EVT_UDECX_USB_DEVICE_D0_ENTRY EvtUdecxUsbDeviceD0Entry;

NTSTATUS EvtUdecxUsbDeviceD0Entry(
    [in] WDFDEVICE UdecxWdfDevice,
    [in] UDECXUSBDEVICE UdecxUsbDevice
)
{...}
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the controller to which the USB device is attached. The client driver initialized this object in a previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE.

## Remarks

The client driver registered the function in a previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#) by supplying a function pointer to its implementation.

In the callback implementation, the client driver for the USB device is expected to perform steps to enter working state.

The power request may be completed asynchronously by returning STATUS\_PENDING, and then later completing it by calling [UdecxUsbDeviceLinkPowerExitComplete](#) with the actual completion code.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

[UdecxUsbDeviceLinkPowerExitComplete](#)

[UdecxUsbDeviceSignalWake](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_DEVICE\_D0\_EXIT callback function (udecxusbdevice.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to send the virtual USB device to a low power state.

## Syntax

C++

```
EVT_UDECX_USB_DEVICE_D0_EXIT EvtUdecxUsbDeviceD0Exit;

NTSTATUS EvtUdecxUsbDeviceD0Exit(
    [in] WDFDEVICE UdecxWdfDevice,
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] UDECX_USB_DEVICE_WAKE_SETTING WakeSetting
)
{...}
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the controller to which the USB device is attached. The client driver initialized this object in a previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

[in] `WakeSetting`

A [UDECX\\_USB\\_DEVICE\\_WAKE\\_SETTING](#)-type value that indicates remote wake capability of the USB device.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE.

## Remarks

The client driver registered the function in a previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#) by supplying a function pointer to its implementation.

In the callback implementation, the client driver for the USB device is expected to perform steps to send the device to a low power state. In this function, the driver can initiate its wake-up from a low link power state, function suspend, or both. To do so, the driver for a USB 2.0 device must call the [UdecxUsbDeviceSignalWake](#) method. USB 3.0 devices must use [UdecxUsbDeviceSignalFunctionWake](#).

The power request may be completed asynchronously by returning STATUS\_PENDING, and then later calling [UdecxUsbDeviceLinkPowerExitComplete](#) with the actual completion code.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#)

[UdecxUsbDeviceLinkPowerExitComplete](#)

[UdecxUsbDeviceSignalWake](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_DEVICE\_DEFAULT\_ENDPOINT\_ADD callback function (udecxusbdevice.h)

Article 10/21/2021

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create the default control endpoint on the virtual USB device.

## Syntax

```
C++  
  
EVT_UDECX_USB_DEVICE_DEFAULT_ENDPOINT_ADD  
EvtUdecxUsbDeviceDefaultEndpointAdd;  
  
NTSTATUS EvtUdecxUsbDeviceDefaultEndpointAdd(  
    [in] UDECXUSBDEVICE UdecxUsbDevice,  
    [in] PUDECXUSBENDPOINT_INIT UdecxEndpointInit  
)  
{...}
```

## Parameters

[in] `UdecxUsbDevice`

A handle to the UDE device object for which the client driver creates the default endpoint. The driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

[in] `UdecxEndpointInit`

A pointer to an `UDECXUSBENDPOINT_INIT` structure that the client driver retrieved in the previous call to [UdecxUsbSimpleEndpointInitAllocate](#).

## Return value

If the operation is successful, the callback function must return `STATUS_SUCCESS`, or another status value for which `NT_SUCCESS(status)` equals TRUE.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbSimpleEndpointInitAllocate](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_DEVICE\_ENDPOINT\_A DD callback function (udecxusbdevice.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function to request the client driver to create a dynamic endpoint on the virtual USB device.

## Syntax

C++

```
EVT_UDECX_USB_DEVICE_ENDPOINT_ADD EvtUdecxUsbDeviceEndpointAdd;

NTSTATUS EvtUdecxUsbDeviceEndpointAdd(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] PUDECX_USB_ENDPOINT_INIT_AND_METADATA EndpointToCreate
)
{...}
```

## Parameters

[in] `UdecxUsbDevice`

A handle to the UDE device object for which the client driver creates an endpoint. The driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

[in] `EndpointToCreate`

A pointer to a [UDECX\\_USB\\_ENDPOINT\\_INIT\\_AND\\_METADATA](#) structure that contains the endpoint descriptor.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE.

## Remarks

The client driver registered this callback function in a previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#) by supplying a function pointer to its

implementation.

In the implementation, the client driver is expected to create the endpoint by calling [UdecxUsbEndpointCreate](#) by using the initialization parameters ([UDECXUSBENDPOINT\\_INIT](#)) passed by the class extension in the [UDECX\\_USB\\_ENDPOINT\\_INIT\\_AND\\_METADATA](#) structure.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointCreate](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_DEVICE\_ENDPOINTS\_C ONFIGURE callback function (udecxusbdevice.h)

Article 10/21/2021

The USB device emulation class extension (UdeCx) invokes this callback function to change the configuration by selecting an alternate setting, disabling current endpoints, or adding dynamic endpoints.

## Syntax

C++

```
EVT_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE
EvtUdecxUsbDeviceEndpointsConfigure;

void EvtUdecxUsbDeviceEndpointsConfigure(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] WDFREQUEST Request,
    [in] PUDECX_ENDPOINTS_CONFIGURE_PARAMS Params
)
{...}
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

[in] `Request`

A handle to a framework request object that represents the request.

[in] `Params`

A pointer to a [UDECX\\_ENDPOINTS\\_CONFIGURE\\_PARAMS](#) structure that describes the configuration options.

## Return value

None

## Remarks

The client driver registered this callback function in a previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#) by supplying a function pointer to its implementation.

The class extension invokes this callback function to request the client driver to configure one or more new endpoints into hardware, and/or informs the driver when one or more existing endpoints is no longer being used.

After creating endpoints, for each new endpoint, the client driver must call [UdecxUsbEndpointSetWdfIoQueue](#) before completing the request.

After releasing endpoints, the client driver should not use framework queue objects associated with the endpoints. The class extension considers those queues as purged to prevent future requests.

The class extension can also request a new configuration value or an alternate setting through this callback.

This call is asynchronous. The client driver must signal completion with status by completing the request passed by the class extension.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointSetWdfIoQueue](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_DEVICE\_SET\_FUNCTION\_SUSPEND\_AND\_WAKE callback function (udecxusbdevice.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function when it gets a request to change the function state of the specified interface of the virtual USB 3.0 device.

## Syntax

C++

```
EVT_UDECX_USB_DEVICE_SET_FUNCTION_SUSPEND_AND_WAKE  
EvtUdecxUsbDeviceSetFunctionSuspendAndWake;  
  
NTSTATUS EvtUdecxUsbDeviceSetFunctionSuspendAndWake(  
    [in] WDFDEVICE UdecxWdfDevice,  
    [in] UDECXUSBDEVICE UdecxUsbDevice,  
    [in] ULONG Interface,  
    [in] UDECX_USB_DEVICE_FUNCTION_POWER FunctionPower  
)  
{...}
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the controller to which the USB device is attached. The client driver initialized this object in a previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver created this object in a previous call to [UdecxUsbDeviceCreate](#).

[in] `Interface`

This value is the `bInterfaceNumber` of the interface that is waking up.

[in] FunctionPower

A [UDECX\\_USB\\_DEVICE\\_FUNCTION\\_POWER](#)-type value that indicates whether the interface can suspend and send wake signal to the host controller.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE.

## Remarks

The client driver registered the function in a previous call to [UdecxUsbDeviceInitSetStateChangeCallbacks](#) by supplying a function pointer to its implementation.

In the callback implementation, the client driver for the USB device is expected to perform steps to enter working state.

This event callback function applies to USB 3.0+ devices. UdeCx invokes this function to notify the client driver of a request to change the power state of a particular function. It also informs the driver whether or not the function can wake from the new state.

The power request may be completed asynchronously by returning STATUS\_PENDING, and then later completing it by calling [UdecxUsbDeviceSetFunctionSuspendAndWakeComplete](#) with the actual completion code.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)

Requirement	Value
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UDECX\_ENDPOINT\_TYPE enumeration (udecxusbdevice.h)

Article02/22/2024

Defines values for endpoint types supported by a virtual USB device.

## Syntax

C++

```
typedef enum _UDECX_ENDPOINT_TYPE {
    UdecxEndpointTypeInvalid,
    UdecxEndpointTypeSimple,
    UdecxEndpointTypeDynamic
} UDECX_ENDPOINT_TYPE, *PUDECX_ENDPOINT_TYPE;
```

## Constants

[+] Expand table

`UdecxEndpointTypeInvalid`

The endpoint is not of a valid type.

`UdecxEndpointTypeSimple`

The endpoint is defined in the first (and only) interface setting of the interface. That device has only one configuration. The client driver creates all endpoints before the device is detected.

`UdecxEndpointTypeDynamic`

The endpoint is dynamically created in the client driver's implementation of the [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback.

## Requirements

[+] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[USB endpoints](#)

[UdecxUsbDeviceInitSetEndpointsType](#)

# UDECX\_ENDPOINTS\_CONFIGURE\_PARAMS structure (udecxusbdevice.h)

Article02/22/2024

Contains the configuration options specified by USB device emulation class extension (UdeCx) to the client driver when the class extension invokes [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#).

## Syntax

C++

```
typedef struct _UDECX_ENDPOINTS_CONFIGURE_PARAMS {
    ULONG                     Size;
    UDECX_ENDPOINTS_CONFIGURE_TYPE ConfigureType;
    UCHAR                     NewConfigurationValue;
    UCHAR                     InterfaceNumber;
    UCHAR                     NewInterfaceSetting;
    ULONG                     EndpointsToConfigureCount;
    UDECXUSBENDPOINT         *EndpointsToConfigure;
    ULONG                     ReleasedEndpointsCount;
    UDECXUSBENDPOINT         *ReleasedEndpoints;
} UDECX_ENDPOINTS_CONFIGURE_PARAMS, *PUDECX_ENDPOINTS_CONFIGURE_PARAMS;
```

## Members

Size

Size of this structure.

ConfigureType

A [UDECX\\_ENDPOINTS\\_CONFIGURE\\_TYPE](#)-typed value that indicates whether the configuration, interface setting, or endpoint must be configured.

NewConfigurationValue

If ConfigureType is [UdecxEndpointsConfigureTypeDeviceConfigurationChange](#), this value is bConfigurationValue of the new configuration descriptor ([USB\\_CONFIGURATION\\_DESCRIPTOR](#)).

InterfaceNumber

If **ConfigureType** is **UdecxEndpointsConfigureTypeInterfaceSettingChange**, this value is **bInterfaceNumber** of the current interface descriptor ([USB\\_INTERFACE\\_DESCRIPTOR](#)).

**NewInterfaceSetting**

If **ConfigureType** is **UdecxEndpointsConfigureTypeInterfaceSettingChange**, this value is **bAlternateSetting** of the interface descriptor ([USB\\_INTERFACE\\_DESCRIPTOR](#)) to set.

**EndpointsToConfigureCount**

The number entries in the array pointed to by *EndpointsToConfigure*. This value indicates number of endpoints that must be configured.

**EndpointsToConfigure**

A pointer to an array of UDECXUSBENDPOINT handles that indicates the endpoint objects to be configured.

A pointer to an array of UDECXUSBENDPOINT handles that indicates the endpoint objects that must be released.

**ReleasedEndpointsCount**

The number entries in the array pointed to by *EndpointsToConfigure*. This value indicates number of endpoints to release.

**ReleasedEndpoints**

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)

# UDECX\_ENDPOINTS\_CONFIGURE\_TYPE enumeration (udecxusbdevice.h)

Article 02/22/2024

Defines values for endpoint configuration options.

## Syntax

C++

```
typedef enum _UDECX_ENDPOINTS_CONFIGURE_TYPE {
    UdecxEndpointsConfigureTypeDeviceInitialize,
    UdecxEndpointsConfigureTypeDeviceConfigurationChange,
    UdecxEndpointsConfigureTypeInterfaceSettingChange,
    UdecxEndpointsConfigureTypeEndpointsReleasedOnly
} UDECX_ENDPOINTS_CONFIGURE_TYPE, *PUDECX_ENDPOINTS_CONFIGURE_TYPE;
```

## Constants

[+] Expand table

`UdecxEndpointsConfigureTypeDeviceInitialize`

Reserved for internal use.

`UdecxEndpointsConfigureTypeDeviceConfigurationChange`

The requested change applies to the USB device configuration.

`UdecxEndpointsConfigureTypeInterfaceSettingChange`

The requested change applies to an alternate setting of a USB interface.

`UdecxEndpointsConfigureTypeEndpointsReleasedOnly`

The requested change applies to an endpoint of an interface setting.

## Requirements

[+] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)

[UDECX\\_ENDPOINTS\\_CONFIGURE\\_PARAMS](#)

# UDECX\_USB\_DEVICE\_CALLBACKS\_INIT function (udecxusbdevice.h)

Article02/22/2024

Initializes a [UDECX\\_USB\\_DEVICE\\_STATE\\_CHANGE\\_CALLBACKS](#) structure before a [UdecxUsbDeviceCreate](#) call.

## Syntax

C++

```
void UDECX_USB_DEVICE_CALLBACKS_INIT(
    [out] PUDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS Callbacks
);
```

## Parameters

[out] Callbacks

A pointer to a [UDECX\\_USB\\_DEVICE\\_STATE\\_CHANGE\\_CALLBACKS](#) structure to initialize.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)

Requirement	Value
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UdecxUsbDeviceCreate](#)

# UDECX\_USB\_DEVICE\_FUNCTION\_POWER enumeration (udecxusbdevice.h)

Article 02/22/2024

Defines values for function wake capability of a virtual USB 3.0 device.

## Syntax

C++

```
typedef enum _UDECX_USB_DEVICE_FUNCTION_POWER {
    UdecxUsbDeviceFunctionNotSuspended,
    UdecxUsbDeviceFunctionSuspendedCannotWake,
    UdecxUsbDeviceFunctionSuspendedCanWake
} UDECX_USB_DEVICE_FUNCTION_POWER, *PUDECX_USB_DEVICE_FUNCTION_POWER;
```

## Constants

[ ] Expand table

`UdecxUsbDeviceFunctionNotSuspended`

The USB interface cannot enter function suspend.

`UdecxUsbDeviceFunctionSuspendedCannotWake`

The USB interface cannot send a wake signal to the host controller.

`UdecxUsbDeviceFunctionSuspendedCanWake`

The USB interface can send a wake signal to the host controller when the function is in suspend state.

## Requirements

[ ] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#)

# UDECX\_USB\_DEVICE\_PLUG\_IN\_OPTIONS structure (udecxusbdevice.h)

Article 02/22/2024

Contains the port numbers to which a virtual USB device is connected. Initialize this structure by calling the [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS\\_INIT](#) method.

## Syntax

C++

```
typedef struct _UDECX_USB_DEVICE_PLUG_IN_OPTIONS {
    ULONG Size;
    ULONG Usb20PortNumber;
    ULONG Usb30PortNumber;
} UDECX_USB_DEVICE_PLUG_IN_OPTIONS, *PUDECX_USB_DEVICE_PLUG_IN_OPTIONS;
```

## Members

Size

The size of this structure.

Usb20PortNumber

The USB 2.0 port number.

Usb30PortNumber

The USB 3.0 port number.

## Requirements

[+] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[UdecxUsbDevicePlugin](#)

# UDECX\_USB\_DEVICE\_PLUG\_IN\_OPTIONS\_INIT function (udecxusbdevice.h)

Article 02/22/2024

Initializes a [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS](#) structure.

## Syntax

C++

```
void UDECX_USB_DEVICE_PLUG_IN_OPTIONS_INIT(
    [out] PUDECX_USB_DEVICE_PLUG_IN_OPTIONS Options
);
```

## Parameters

[out] Options

A pointer to a [UDECX\\_USB\\_DEVICE\\_PLUG\\_IN\\_OPTIONS](#) structure to initialize.

## Return value

None

## Remarks

The method initializes **Usb20PortNumber** and **Usb30PortNumber** to 0. This indicates a request for automatic port number selection.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UdecxUsbDevicePlugin](#)

# UDECX\_USB\_DEVICE\_SPEED enumeration (udecxusbdevice.h)

Article 02/22/2024

Defines values for USB device speeds.

## Syntax

C++

```
typedef enum _UDECX_USB_DEVICE_SPEED {
    UdecxUsbLowSpeed,
    UdecxUsbFullSpeed,
    UdecxUsbHighSpeed,
    UdecxUsbSuperSpeed
} UDECX_USB_DEVICE_SPEED, *PUDECX_USB_DEVICE_SPEED;
```

## Constants

[ ] Expand table

	<b>UdecxUsbLowSpeed</b>
	Indicates a low-speed USB 1.1-compliant device.
	<b>UdecxUsbFullSpeed</b>
	Indicates a full-speed USB 1.1-compliant device.
	<b>UdecxUsbHighSpeed</b>
	Indicates a high-speed USB 2.0-compliant device.
	<b>UdecxUsbSuperSpeed</b>
	Indicates a SuperSpeed USB 3.0-compliant device.

## Requirements

[ ] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[UdecxUsbDeviceInitSetSpeed](#)

# UDECX\_USB\_DEVICE\_STATE\_CHANGE\_C ALLBACKS structure (udecxusbdevice.h)

Article02/22/2024

Initializes a UDECX\_USB\_DEVICE\_STATE\_CHANGE\_CALLBACKS structure with pointers to callback functions that are implemented by a UDE client for a virtual USB device.

## Syntax

C++

```
typedef struct _UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS {
    ULONG Size;
    PFN_UDECX_USB_DEVICE_D0_ENTRY EvtUsbDeviceLinkPowerEntry;
    PFN_UDECX_USB_DEVICE_D0_EXIT EvtUsbDeviceLinkPowerExit;
    PFN_UDECX_USB_DEVICE_SET_FUNCTION_SUSPEND_AND_WAKE
    EvtUsbDeviceSetFunctionSuspendAndWake;
    PFN_UDECX_USB_DEVICE_POST_ENUMERATION_RESET EvtUsbDeviceReset;
    PFN_UDECX_USB_DEVICE_DEFAULT_ENDPOINT_ADD
    EvtUsbDeviceDefaultEndpointAdd;
    PFN_UDECX_USB_DEVICE_ENDPOINT_ADD
    EvtUsbDeviceEndpointAdd;
    PFN_UDECX_USB_DEVICE_ENDPOINTS_CONFIGURE
    EvtUsbDeviceEndpointsConfigure;
} UDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS,
*PUDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS;
```

## Members

Size

The size of this structure.

EvtUsbDeviceLinkPowerEntry

A pointer to an [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#) callback function implemented by a UDE client driver.

EvtUsbDeviceLinkPowerExit

A pointer to an [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#) callback function implemented by a UDE client driver.

`EvtUsbDeviceSetFunctionSuspendAndWake`

`EvtUsbDeviceReset`

`EvtUsbDeviceDefaultEndpointAdd`

A pointer to an [EVT\\_UDECX\\_USB\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) callback function implemented by a UDE client driver.

`EvtUsbDeviceEndpointAdd`

A pointer to an [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINT\\_ADD](#) callback function implemented by a UDE client driver.

`EvtUsbDeviceEndpointsConfigure`

A pointer to an [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback function implemented by a UDE client driver.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[UdecxUsbDeviceInitSetStateChangeCallbacks](#)

# UDECX\_USB\_DEVICE\_WAKE\_SETTING enumeration (udecxusbdevice.h)

Article 02/22/2024

Defines values for remote wake capability of a virtual USB device.

## Syntax

C++

```
typedef enum _UDECX_USB_DEVICE_WAKE_SETTING {
    UdecxUsbDeviceWakeDisabled,
    UdecxUsbDeviceWakeEnabled,
    UdecxUsbDeviceWakeNotApplicable
} UDECX_USB_DEVICE_WAKE_SETTING, *PUDECX_USB_DEVICE_WAKE_SETTING;
```

## Constants

[+] Expand table

`UdecxUsbDeviceWakeDisabled`

The USB device cannot send a wake signal to the host controller.

`UdecxUsbDeviceWakeEnabled`

The USB device can send a wake signal to the host controller.

`UdecxUsbDeviceWakeNotApplicable`

This value is used only if the USB device is a SuperSpeed device.

## Requirements

[+] Expand table

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

# UDECX\_USB\_ENDPOINT\_INIT\_AND\_METADATA structure (udecxusbdevice.h)

Article02/22/2024

Contains the descriptors supported by an endpoint of a virtual USB device.

## Syntax

C++

```
typedef struct _UDECX_USB_ENDPOINT_INIT_AND_METADATA {
    PUDECXUSBENDPOINT_INIT UdecxUsbEndpointInit;
    ULONG EndpointDescriptorBufferLength;
    PUSB_ENDPOINT_DESCRIPTOR EndpointDescriptor;
    PUSB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR
    SuperSpeedEndpointCompanionDescriptor;
} UDECX_USB_ENDPOINT_INIT_AND_METADATA,
*PUDECX_USB_ENDPOINT_INIT_AND_METADATA;
```

## Members

`UdecxUsbEndpointInit`

A pointer to a **UDECXUSBDEVICE\_INIT** structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

`EndpointDescriptorBufferLength`

The length of the endpoint descriptor.

`EndpointDescriptor`

Required. A buffer containing the endpoint descriptor. The descriptor is described in a [USB\\_ENDPOINT\\_DESCRIPTOR](#) structure.

`SuperSpeedEndpointCompanionDescriptor`

Optional. A USB-defined SuperSpeed Endpoint Companion descriptor. For more information, see section 9.6.7 and Table 9-20 in the official USB 3.0 specification. The

descriptor is described in a [USB\\_SUPERSPEED\\_ENDPOINT\\_COMPANION\\_DESCRIPTOR](#) structure.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINT\\_ADD](#)

# UdecxUsbDeviceCreate function (udecxusbdevice.h)

Article 02/22/2024

Creates a USB Device Emulation (UDE) device object.

## Syntax

C++

```
NTSTATUS UdecxUsbDeviceCreate(
    [in, out]     PUDECXUSBDEVICE_INIT      *UdecxUsbDeviceInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]          UDECXUSBDEVICE           *UdecxUsbDevice
);
```

## Parameters

[in, out] `UdecxUsbDeviceInit`

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in, optional] `Attributes`

A pointer to a caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that specifies attributes for the USB device object.

[out] `UdecxUsbDevice`

A pointer to a variable that receives a handle to the new UDE device object that represents the virtual USB device.

## Return value

The method returns `STATUS_SUCCESS` if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitAddDescriptor function (udecxusbdevice.h)

Article 10/21/2021

Adds a USB descriptor to the initialization parameters used to create a virtual USB device.

## Syntax

C++

```
NTSTATUS UdecxUsbDeviceInitAddDescriptor(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit,
    [in]     P UCHAR             Descriptor,
    [in]     USHORT            DescriptorLength
);
```

## Parameters

[in, out] `UdecxUsbDeviceInit`

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] `Descriptor`

A caller-allocated buffer that contains the USB descriptor to add to the device.

[in] `DescriptorLength`

The length of the descriptor buffer.

## Return value

The method returns `STATUS_SUCCESS` if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Standard USB descriptors](#)

[UdecxUsbDeviceInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitAddDescriptorWithIndex function (udecxusbdevice.h)

Article 02/22/2024

Adds a USB descriptor to the initialization parameters used to create a virtual USB device.

## Syntax

C++

```
NTSTATUS UdecxUsbDeviceInitAddDescriptorWithIndex(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit,
    [in]     P UCHAR             Descriptor,
    [in]     USHORT            DescriptorLength,
    [in]     UCHAR             DescriptorIndex
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] Descriptor

A caller-allocated buffer that contains the USB descriptor to add to the device.

[in] DescriptorLength

The length of the descriptor buffer.

[in] DescriptorIndex

The index of the descriptor.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[USB String Descriptors](#)

[UdecxUsbDeviceInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitAddStringDescriptor function (udecxusbdevice.h)

Article 02/22/2024

Adds a USB string descriptor to the initialization parameters used to create a virtual USB device.

## Syntax

C++

```
NTSTATUS UdecxUsbDeviceInitAddStringDescriptor(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit,
    [in]      PCUNICODE_STRING     String,
    [in]      UCHAR                DescriptorIndex,
    [in]      USHORT               LanguageId
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] String

A Unicode string that contains the USB string descriptor to add to the device.

[in] DescriptorIndex

The index of the descriptor.

[in] LanguageId

The language identifier of the string. The client driver must define constants for the language support, such as:

```
const USHORT US_ENGLISH = 0x409;
```

# Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[USB String Descriptors](#)

[UdecxUsbDeviceInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitAddStringDescriptorRaw function (udecxusbdevice.h)

Article02/22/2024

Adds a USB string descriptor to the initialization parameters used to create a virtual USB device.

## Syntax

C++

```
NTSTATUS UdecxUsbDeviceInitAddStringDescriptorRaw(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit,
    [in]     PUCHAR             Descriptor,
    [in]     USHORT            DescriptorLength,
    [in]     UCHAR             DescriptorIndex,
    [in]     USHORT            LanguageId
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] Descriptor

A caller-allocated buffer that contains the USB descriptor to add to the device.

[in] DescriptorLength

The length of the descriptor buffer.

[in] DescriptorIndex

The index of the descriptor.

[in] LanguageId

The language identifier of the string. The client driver must define constants for the language support, such as:

```
const USHORT US_ENGLISH = 0x409;
```

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[USB String Descriptors](#)

[UdecxUsbDeviceInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitAllocate function (udecxusbdevice.h)

Article 02/22/2024

Allocates memory for a **UDECXUSBDEVICE\_INIT** structure that is used to initialize a virtual USB device.

## Syntax

C++

```
PUDECXUSBDEVICE_INIT UdecxUsbDeviceInitAllocate(  
    [in] WDFDEVICE UdecxWdfDevice  
)
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the a USB device. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

## Return value

This method returns a pointer to an opaque **UDECXUSBDEVICE\_INIT** that contains the initialization parameters. The structure is allocated by the USB device emulation class extension (UdeCx).

## Remarks

The UDE client driver calls this method to allocate parameters for the virtual device that is created by a subsequent call to [UdecxUsbDeviceCreate](#). If the device is not created or the driver is finished using the resources, the driver must free the resources by calling [UdecxUsbDeviceInitFree](#).

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitFree function (udecxusbdevice.h)

Article02/22/2024

Releases the resources that were allocated by the [UdecxUsbDeviceInitAllocate](#) call.

## Syntax

C++

```
void UdecxUsbDeviceInitFree(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit
);
```

## Parameters

[in, out] `UdecxUsbDeviceInit`

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Requirement	Value
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitSetEndpointsType function (udecxusbdevice.h)

Article 02/22/2024

Indicates the type of endpoint (simple or dynamic) in the initialization parameters that the client driver uses to create the virtual USB device.

## Syntax

C++

```
void UdecxUsbDeviceInitSetEndpointsType(
    [in, out] PUDECXUSBDEVICE_INIT UdecxUsbDeviceInit,
    [in]      UDECX_ENDPOINT_TYPE  UdecxEndpointType
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] UdecxEndpointType

A [UDECX\\_ENDPOINT\\_TYPE](#)-type value that indicates the type of USB endpoint.

## Return value

None

## Remarks

Before creating the virtual USB device, the client driver must indicate the type of endpoint it supports. It can support one of two types (defined in [UDECX\\_ENDPOINT\\_TYPE](#)):

- Simple endpoint-The client driver creates all endpoint objects before plugging in the device. The device must have only one configuration and one interface setting per interface.
- Dynamic endpoint-The client creates endpoint objects in the [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#) callback function. The USB device emulation class extension (UdeCx) invokes the driver's implementation when it gets a request to add or configure endpoints.

The [UdecxUsbDeviceInit](#) is an opaque structure that contains pointers to callback functions related to endpoints. If the client driver supports dynamic endpoints, then these callback functions must be implemented by the driver:

- [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)
- [EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINT\\_ADD](#)
- [EVT\\_UDECX\\_USB\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)

Before calling this method, the client driver must have set those pointers by calling [UdecxUsbDeviceInitSetStateChangeCallbacks](#).

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_ENDPOINTS\\_CONFIGURE](#)

[USB endpoints](#)

[UdecxUsbDeviceInitAllocate](#)

[UdecxUsbDeviceInitSetStateChangeCallbacks](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitSetSpeed function (udecxusbdevice.h)

Article02/22/2024

Sets the USB speed of the virtual USB device to create.

## Syntax

C++

```
void UdecxUsbDeviceInitSetSpeed(
    [in, out] PUDECXUSBDEVICE_INIT    UdecxUsbDeviceInit,
    [in]      UDECX_USB_DEVICE_SPEED UsbDeviceSpeed
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] UsbDeviceSpeed

A [UDECX\\_USB\\_DEVICE\\_SPEED](#)-type value that indicates the USB speed to set.

## Return value

None

## Remarks

After the client driver sets the USB speed of the device, it only operates in that speed. The speed also determines the kind of port to which the device can connect. For example, a USB SuperSpeed device cannot connect to a USB 2.0 port.

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceInitSetStateChangeCallbacks function (udecxusbdevice.h)

Article 02/22/2024

Initializes a WDF-allocated structure with pointers to callback functions.

## Syntax

C++

```
void UdecxUsbDeviceInitSetStateChangeCallbacks(
    [in, out] PUDECXUSBDEVICE_INIT                  UdecxUsbDeviceInit,
    [in]      PUDECX_USB_DEVICE_STATE_CHANGE_CALLBACKS Callbacks
);
```

## Parameters

[in, out] UdecxUsbDeviceInit

A pointer to a WDF-allocated structure that contains initialization parameters for the virtual USB device. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceInitAllocate](#).

[in] Callbacks

A pointer to a [UDECX\\_USB\\_DEVICE\\_STATE\\_CHANGE\\_CALLBACKS](#) structure that contains pointers to callback functions implemented by the client driver.

## Return value

None

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbDeviceInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceLinkPowerEntryComplete function (udecxusbdevice.h)

Article 02/22/2024

Completes an asynchronous request for bringing the device out of a low power state.

## Syntax

C++

```
void UdecxUsbDeviceLinkPowerEntryComplete(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] NTSTATUS       CompletionStatus
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

[in] `CompletionStatus`

An appropriate [NTSTATUS](#) error code that indicates the success or failure of the asynchronous operation.

## Return value

None

## Remarks

When the USB device emulation class extension (UdeCx) gets a request to bring the device from low power state and enter working state, it invokes the client driver's implementation of the [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#) callback function.

After the client driver has performed the necessary steps for bringing the virtual USB device to working state, the driver calls this method to notify the class extension that it

has completed the power request.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_ENTRY](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceLinkPowerExitComplete function (udecxusbdevice.h)

Article 02/22/2024

Completes an asynchronous request for sending the device to a low power state.

## Syntax

C++

```
void UdecxUsbDeviceLinkPowerExitComplete(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] NTSTATUS       CompletionStatus
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

[in] `CompletionStatus`

An appropriate [NTSTATUS](#) error code that indicates the success or failure of the asynchronous operation.

## Return value

None

## Remarks

When the USB device emulation class extension (UdeCx) gets a request to send the device to a low power state, it invokes the client driver's implementation of the [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#) callback function.

After the client driver has performed the necessary steps for sending the virtual USB device to low power state, the driver calls this method to notify the class extension that

it has completed the power request.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

[Write a UDE client driver](#)

# UdecxUsbDevicePlugIn function (udecxusbdevice.h)

Article02/22/2024

Notifies the USB device emulation class extension (UdeCx) that the USB device has been plugged in the specified port.

## Syntax

C++

```
NTSTATUS UdecxUsbDevicePlugIn(
    [in] UDECXUSBDEVICE                 UdecxUsbDevice,
    [in] PUDECX_USB_DEVICE_PLUG_IN_OPTIONS Options
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

[in] `Options`

A `UDECX_USB_DEVICE_PLUG_IN_OPTIONS`-type value that indicates the port to which the device is plugged. At most one of `Usb20PortNumber`, `Usb30PortNumber` can be non-zero. `NULL` disables plug-in options (use defaults).

## Return value

The method returns `STATUS_SUCCESS` if the operation succeeds. Otherwise, this method might return an appropriate `NTSTATUS` error code.

## Remarks

After the client driver calls this method, the class extension sends I/O requests and invokes callback functions on the endpoints and the device.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDevicePlugOutAndDelete function (udecxusbdevice.h)

Article 10/21/2021

Disconnects the virtual USB device.

## Syntax

C++

```
NTSTATUS UdecxUsbDevicePlugOutAndDelete(
    [in] UDECXUSBDEVICE UdecxUsbDevice
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

If the USB device needs to be removed at runtime, the client driver can call this method to indicate a disconnect event. After this call completes, the client driver can no longer use the device specified by the `UdecxUsbDevice` parameter; it must create another device by calling [UdecxUsbDeviceCreate](#).

## Requirements

Minimum supported client

Windows 10

<b>Minimum supported server</b>	Windows Server 2016
<b>Target Platform</b>	Windows
<b>Minimum KMDF version</b>	1.15
<b>Header</b>	udecxusbdevice.h (include Udecx.h)
<b>Library</b>	Udecxstub.lib
<b>IRQL</b>	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceSetFunctionSuspendAndWakeComplete function (udecxusbdevice.h)

Article02/22/2024

Completes an asynchronous request for changing the power state of a particular function of a virtual USB 3.0 device.

## Syntax

C++

```
void UdecxUsbDeviceSetFunctionSuspendAndWakeComplete(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] NTSTATUS         CompletionStatus
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

[in] `CompletionStatus`

An appropriate [NTSTATUS](#) error code that indicates the success or failure of the asynchronous operation.

## Return value

None

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceSignalFunctionWake function (udecxusbdevice.h)

Article02/22/2024

Initiates wake up of the specified function from a low power state. This applies to virtual USB 3.0 devices.

## Syntax

C++

```
void UdecxUsbDeviceSignalFunctionWake(
    [in] UDECXUSBDEVICE UdecxUsbDevice,
    [in] ULONG           Interface
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

[in] `Interface`

This value is the `bInterfaceNumber` of the interface that is waking up.

## Return value

None

## Remarks

The client driver for the device must have enabled wake capability in the most recent [EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#) call.

If the device is in a low power state, or going to such a state, this call also wakes up the entire device.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_SET\\_FUNCTION\\_SUSPEND\\_AND\\_WAKE](#)

[Write a UDE client driver](#)

# UdecxUsbDeviceSignalWake function (udecxusbdevice.h)

Article02/22/2024

Initiates wake up from a low link power state for a virtual USB 2.0 device.

## Syntax

C++

```
void UdecxUsbDeviceSignalWake(
    [in] UDECXUSBDEVICE UdecxUsbDevice
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

## Return value

None

## Remarks

The client driver for the device must have enabled wake capability in the most recent [EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#) call.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_DEVICE\\_D0\\_EXIT](#)

[Write a UDE client driver](#)

# udecxusbendpoint.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

udecxusbendpoint.h contains the following programming interfaces:

## Functions

<a href="#">UDECX_USB_ENDPOINT_CALLBACKS_INIT</a>
Initializes a UDECX_USB_ENDPOINT_CALLBACKS structure before a UdecxUsbEndpointCreate call.
<a href="#">UdecxUsbEndpointCreate</a>
Creates a UDE endpoint object.
<a href="#">UdecxUsbEndpointInitFree</a>
Release the resources that were allocated by the UdecxUsbSimpleEndpointInitAllocate call.
<a href="#">UdecxUsbEndpointInitSetCallbacks</a>
Sets pointers to UDE client driver-implemented callback functions in the initialization parameters of the simple endpoint to create.
<a href="#">UdecxUsbEndpointInitSetEndpointAddress</a>
Sets the address of the endpoint in the initialization parameters of the simple endpoint to create.
<a href="#">UdecxUsbEndpointPurgeComplete</a>
Completes an asynchronous request for canceling all I/O requests queued to the specified endpoint.
<a href="#">UdecxUsbEndpointSetWdfIoQueue</a>
Sets a framework queue object with a UDE endpoint.
<a href="#">UdecxUsbSimpleEndpointInitAllocate</a>
Allocates memory for an initialization structure that is used to create a simple endpoint for the specified virtual USB device.

# Callback functions

## [EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#)

The USB device emulation class extension (UdeCx) invokes this callback function to stop queuing I/O requests to the endpoint's queue and cancel unprocessed requests.

## [EVT\\_UDECX\\_USB\\_ENDPOINT\\_RESET](#)

The USB device emulation class extension (UdeCx) invokes this callback function to reset an endpoint of the virtual USB device.

## [EVT\\_UDECX\\_USB\\_ENDPOINT\\_START](#)

The USB device emulation class extension (UdeCx) invokes this callback function to start processing I/O requests on the specified endpoint of the virtual USB device.

# Structures

## [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS](#)

Contains function pointers to endpoint callback functions implemented by the UDE client driver. Initialize this structure by calling [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS\\_INIT](#).

# EVT\_UDECX\_USB\_ENDPOINT\_PURGE callback function (udecxusbendpoint.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function to stop queuing I/O requests to the endpoint's queue and cancel unprocessed requests.

## Syntax

C++

```
EVT_UDECX_USB_ENDPOINT_PURGE EvtUdecxUsbEndpointPurge;

void EvtUdecxUsbEndpointPurge(
    [in] UDECXUSBENDPOINT UdecxUsbEndpoint
)
{...}
```

## Parameters

[in] `UdecxUsbEndpoint`

A handle to a UDE endpoint object that represents the endpoint for which I/O requests must be canceled. The client driver retrieved this pointer in the previous call to [UdecxUsbEndpointCreate](#).

## Return value

None

## Remarks

The client driver registered this callback function in a previous call to [UdecxUsbEndpointInitSetCallbacks](#) by supplying a function pointer to its implementation.

In the implementation, the client driver is required to ensure all I/O forwarded from the endpoint's queue has been completed, and that newly forwarded I/O request fail, until UdeCx invokes [EVT\\_UDECX\\_USB\\_ENDPOINT\\_START](#). Typically, those tasks are achieved

by calling [WdfIoQueuePurge](#). This call is asynchronous and the client river must call [UdecxUsbEndpointPurgeComplete](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Managing I/O Queues](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_ENDPOINT\_RESET callback function (udecxusbendpoint.h)

Article02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function to reset an endpoint of the virtual USB device.

## Syntax

C++

```
EVT_UDECX_USB_ENDPOINT_RESET EvtUdecxUsbEndpointReset;

void EvtUdecxUsbEndpointReset(
    [in] UDECXUSBENDPOINT UdecxUsbEndpoint,
    [in] WDFREQUEST Request
)
{...}
```

## Parameters

[in] `UdecxUsbEndpoint`

A handle to a UDE endpoint object that represents the endpoint to reset. The client driver retrieved this pointer in the previous call to [UdecxUsbEndpointCreate](#).

[in] `Request`

A handle to a framework request object that represents the request to reset the endpoint.

## Return value

None

## Remarks

The client driver registered this callback function in a previous call to [UdecxUsbEndpointInitSetCallbacks](#) by supplying a function pointer to its implementation.

The reset request clears the error condition in the endpoint that causes failed I/O transfers. At that time, UdeCx can invoke the *EVT\_UDECX\_USB\_ENDPOINT\_RESET* callback function. That call is asynchronous. The client driver completes the request and signals completion with status by calling [WdfRequestCompleteWithInformation](#) method . (this is the only way the UDECX client uses the request parameter).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

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

[Managing I/O Queues](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_USB\_ENDPOINT\_START callback function (udecxusbendpoint.h)

Article 02/22/2024

The USB device emulation class extension (UdeCx) invokes this callback function to start processing I/O requests on the specified endpoint of the virtual USB device.

## Syntax

C++

```
EVT_UDECX_USB_ENDPOINT_START EvtUdecxUsbEndpointStart;

void EvtUdecxUsbEndpointStart(
    [in] UDECXUSBENDPOINT UdecxUsbEndpoint
)
{...}
```

## Parameters

[in] `UdecxUsbEndpoint`

A handle to a UDE endpoint object that represents the endpoint which can start receiving I/O requests. The client driver retrieved this pointer in the previous call to [UdecxUsbEndpointCreate](#).

## Return value

None

## Remarks

The client driver registered this callback function in a previous call to [UdecxUsbEndpointInitSetCallbacks](#) by supplying a function pointer to its implementation.

After the client driver creates an endpoint, it does not automatically start receiving I/O requests. When UdeCx is ready to forward those request for processing, it invokes the client driver's *EVT\_UDECX\_USB\_ENDPOINT\_START* function and the client driver can

begin processing I/O on the endpoint's queue, and on any queues that receive forwarded I/O for the endpoint. This callback returns the endpoint to a state of processing I/O after an [EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#) callback has and completed.

## Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#)

[Managing I/O Queues](#)

[Write a UDE client driver](#)

# UDECX\_USB\_ENDPOINT\_CALLBACKS structure (udecxusbendpoint.h)

Article 04/01/2021

Contains function pointers to endpoint callback functions implemented by the UDE client driver. Initialize this structure by calling [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS\\_INIT](#).

## Syntax

C++

```
typedef struct _UDECX_USB_ENDPOINT_CALLBACKS {
    ULONG Size;
    PFN_UDECX_USB_ENDPOINT_RESET EvtUsbEndpointReset;
    PFN_UDECX_USB_ENDPOINT_START EvtUsbEndpointStart;
    PFN_UDECX_USB_ENDPOINT_PURGE EvtUsbEndpointPurge;
} UDECX_USB_ENDPOINT_CALLBACKS, *PUDECX_USB_ENDPOINT_CALLBACKS;
```

## Members

Size

The size of this structure.

EvtUsbEndpointReset

Required. A pointer to an [EVT\\_UDECX\\_USB\\_ENDPOINT\\_RESET](#) callback function implemented by a UDE client driver.

EvtUsbEndpointStart

Optional. A pointer to an [EVT\\_UDECX\\_USB\\_ENDPOINT\\_START](#) callback function implemented by a UDE client driver.

EvtUsbEndpointPurge

Optional. A pointer to an [EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#) callback function implemented by a UDE client driver.

## Requirements

Header	udecxusbendpoint.h (include Udecx.h)
--------	--------------------------------------

## See also

[UdecxUsbEndpointCreate](#)

[UdecxUsbEndpointInitSetCallbacks](#)

# UDECX\_USB\_ENDPOINT\_CALLBACKS\_INIT function (udecxusbendpoint.h)

Article02/22/2024

Initializes a [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS](#) structure before a [UdecxUsbEndpointCreate](#) call.

## Syntax

C++

```
void UDECX_USB_ENDPOINT_CALLBACKS_INIT(
    [out] PUDECX_USB_ENDPOINT_CALLBACKS Callbacks,
        PFN_UDECX_USB_ENDPOINT_RESET EvtUsbEndpointReset
);
```

## Parameters

[out] `Callbacks`

A pointer to a [UDECX\\_USB\\_ENDPOINT\\_CALLBACKS](#) to initialize.

`EvtUsbEndpointReset`

Pointer to an [EVT\\_UDECX\\_USB\\_ENDPOINT\\_RESET](#) callback function.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UdecxUsbEndpointCreate](#)

# UdecxUsbEndpointCreate function (udecxusbendpoint.h)

Article02/22/2024

Creates a UDE endpoint object.

## Syntax

C++

```
NTSTATUS UdecxUsbEndpointCreate(
    PUDECXUSBENDPOINT_INIT *EndpointInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]           UDECXUSBENDPOINT      *UdecxUsbEndpoint
);
```

## Parameters

EndpointInit

A pointer to an **UDECXUSBENDPOINT\_INIT** structure that the client driver retrieved in the previous call to [UdecxUsbSimpleEndpointInitAllocate](#).

[in, optional] Attributes

A pointer to a caller-allocated **WDF\_OBJECT\_ATTRIBUTES** structure that specifies attributes for the USB device object.

[out] UdecxUsbEndpoint

A pointer to a variable that receives a handle to the new UDE endpoint object that represents the simple endpoint on the USB device.

## Return value

The method returns **STATUS\_SUCCESS** if the operation succeeds. Otherwise, this method might return an appropriate **NTSTATUS** error code.

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbSimpleEndpointInitAllocate](#)

[Write a UDE client driver](#)

# UdecxUsbEndpointInitFree function (udecxusbendpoint.h)

Article02/22/2024

Release the resources that were allocated by the [UdecxUsbSimpleEndpointInitAllocate](#) call.

## Syntax

C++

```
void UdecxUsbEndpointInitFree(
    [in] PUDECXUSBENDPOINT_INIT Init
);
```

## Parameters

[in] Init

A pointer to an **UDECXUSBENDPOINT\_INIT** structure that the client driver retrieved in the previous call to [UdecxUsbSimpleEndpointInitAllocate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Requirement	Value
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointCreate](#)

[Write a UDE client driver](#)

# UdecxUsbEndpointInitSetCallbacks function (udecxusbendpoint.h)

Article02/22/2024

Sets pointers to UDE client driver-implemented callback functions in the initialization parameters of the simple endpoint to create.

## Syntax

C++

```
void UdecxUsbEndpointInitSetCallbacks(
    PUDECXUSBENDPOINT_INIT          UdecxUsbEndpointInit,
    [in] PUDECX_USB_ENDPOINT_CALLBACKS EndpointCallbacks
);
```

## Parameters

`UdecxUsbEndpointInit`

A pointer to an `UDECXUSBENDPOINT_INIT` structure that the client driver retrieved in the previous call to [UdecxUsbSimpleEndpointInitAllocate](#).

`[in] EndpointCallbacks`

A pointer to `UDECX_USB_ENDPOINT_CALLBACKS` that contains function pointers to event callback functions implemented by the UDE client driver.

## Return value

None

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10

Requirement	Value
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointCreate](#)

[Write a UDE client driver](#)

# UdecxUsbEndpointInitSetEndpointAddress function (udecxusbendpoint.h)

Article 02/22/2024

Sets the address of the endpoint in the initialization parameters of the simple endpoint to create.

## Syntax

C++

```
void UdecxUsbEndpointInitSetEndpointAddress(
    [in, out] PUDECXUSBENDPOINT_INIT Init,
    [in]      UCHAR             EndpointAddress
);
```

## Parameters

[in, out] Init

A pointer to an **UDECXUSBENDPOINT\_INIT** structure that the client driver retrieved in the previous call to [UdecxUsbSimpleEndpointInitAllocate](#).

[in] EndpointAddress

Specifies the USB-defined endpoint address. The four low-order bits specify the endpoint number. The high-order bit specifies the direction of data flow on this endpoint: 1 for in, 0 for out.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointCreate](#)

[Write a UDE client driver](#)

# UdecxUsbEndpointPurgeComplete function (udecxusbendpoint.h)

Article02/22/2024

Completes an asynchronous request for canceling all I/O requests queued to the specified endpoint.

## Syntax

C++

```
void UdecxUsbEndpointPurgeComplete(
    [in] UDECXUSBENDPOINT UdecxUsbEndpoint
);
```

## Parameters

[in] `UdecxUsbEndpoint`

A handle to a UDE endpoint object. The client driver retrieved this pointer in the previous call to [UdecxUsbEndpointCreate](#).

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Requirement	Value
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_USB\\_ENDPOINT\\_PURGE](#)

[Write a UDE client driver](#)

# UdecxUsbEndpointSetWdfIoQueue function (udecxusbendpoint.h)

Article 10/21/2021

Sets a framework queue object with a UDE endpoint.

## Syntax

C++

```
void UdecxUsbEndpointSetWdfIoQueue(
    [in] UDECXUSBENDPOINT UdecxUsbEndpoint,
    [in] WDFQUEUE         WdfQueue
);
```

## Parameters

[in] `UdecxUsbEndpoint`

A handle to a UDE endpoint object. The client driver retrieved this pointer in the previous call to [UdecxUsbEndpointCreate](#).

[in] `WdfQueue`

A handle to a framework queue object that will handle requests sent to the endpoint. The client driver retrieved this pointer in the previous call to [WdfIoQueueCreate](#).

## Return value

None

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows

Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Creating I/O Queues](#)

[Write a UDE client driver](#)

# UdecxUsbSimpleEndpointInitAllocate function (udecxusbendpoint.h)

Article 02/22/2024

Allocates memory for an initialization structure that is used to create a simple endpoint for the specified virtual USB device.

## Syntax

C++

```
PUDECXUSBENDPOINT_INIT UdecxUsbSimpleEndpointInitAllocate(  
    [in] UDECXUSBDEVICE UdecxUsbDevice  
);
```

## Parameters

[in] `UdecxUsbDevice`

A handle to UDE device object. The client driver retrieved this pointer in the previous call to [UdecxUsbDeviceCreate](#).

## Return value

This method returns a pointer to an opaque `UDECXUSBENDPOINT_INIT` structure that contains the initialization parameters. The structure is allocated by the USB device emulation class extension (UdeCx).

## Remarks

The UDE client driver calls this method to allocate parameters for a simple endpoint that is created by a subsequent call to [UdecxUsbEndpointCreate](#). If the device is not created or the driver is finished using the resources, the driver must free the resources by calling [UdecxUsbEndpointInitFree](#).

The only valid time to create simple endpoints is after creating a the UDE device object and before calling [UdecxUsbDevicePlugIn](#) on the device.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxusbendpoint.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[UdecxUsbEndpointCreate](#)

[Write a UDE client driver](#)

# udecxwdfdevice.h header

Article01/23/2023

This header is used 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).

Do not include this header directly. Instead include Udecx.h.

For more information, see:

- [Universal Serial Bus \(USB\)](#)
- [Developing Windows drivers for emulated USB devices \(UDE\)](#)

udecxwdfdevice.h contains the following programming interfaces:

## Functions

<a href="#">UDECX_WDF_DEVICE_CONFIG_INIT</a>
Initializes a UDECX_WDF_DEVICE_CONFIG structure.
<a href="#">UdecxInitializeWdfDeviceInit</a>
UdecxInitializeWdfDeviceInit initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UdecxWdfDeviceAddUsbDeviceEmulation</a>
Initializes a framework device object to support operations related to a host controller and a virtual USB device attached to the controller.
<a href="#">UdecxWdfDeviceNeedsReset</a>
Informs the USB device emulation class extension (UdeCx) that the device needs a reset operation.
<a href="#">UdecxWdfDeviceResetComplete</a>
Informs the USB device emulation class extension (UdeCx) that the reset operation on the specified controller has completed.

## [UdecxWdfDeviceTryHandleUserIoctl](#)

Attempts to handle an IOCTL request sent by a user-mode software.

# Callback functions

## [EVT\\_UDECX\\_WDF\\_DEVICE\\_QUERY\\_USB\\_CAPABILITY](#)

The UDE client driver's implementation to determine the capabilities that are supported by the emulated USB host controller.

## [EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#)

The UDE client driver's implementation to reset the emulated host controller or the devices attached to it.

# Structures

## [UDECX\\_WDF\\_DEVICE\\_CONFIG](#)

Contains pointers to event callback functions implemented by the UDE client driver for a USB host controller. Initialize this structure by calling [UDECX\\_WDF\\_DEVICE\\_CONFIG\\_INIT](#).

# Enumerations

## [UDECX\\_WDF\\_DEVICE\\_RESET\\_ACTION](#)

Defines values that indicate the types of reset operation supported by an emulated USB host controller.

## [UDECX\\_WDF\\_DEVICE\\_RESET\\_TYPE](#)

Defines values that indicates the type of reset for a UDE device.

# EVT\_UDECX\_WDF\_DEVICE\_QUERY\_USB\_CAPABILITY callback function (udecxwdfdevice.h)

Article10/21/2021

The UDE client driver's implementation to determine the capabilities that are supported by the emulated USB host controller.

## Syntax

```
C++  
  
EVT_UDECX_WDF_DEVICE_QUERY_USB_CAPABILITY  
EvtUdecxWdfDeviceQueryUsbCapability;  
  
NTSTATUS EvtUdecxWdfDeviceQueryUsbCapability(  
    [in]           WDFDEVICE UdecxWdfDevice,  
    [in]           PGUID CapabilityType,  
    [in]           ULONG OutputBufferLength,  
    [out, optional] PVOID OutputBuffer,  
    [out]          PULONG ResultLength  
)  
{...}
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the controller. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

[in] `CapabilityType`

Pointer to a GUID specifying the requested capability. The possible *PGUID* values are as follows:

- `GUID_USB_CAPABILITY_CHAINED_MDLS`
- `GUID_USB_CAPABILITY_SELECTIVE_SUSPEND`
- `GUID_USB_CAPABILITY_FUNCTION_SUSPEND`
- `GUID_USB_CAPABILITY_DEVICE_CONNECTION_HIGH_SPEED_COMPATIBLE`
- `GUID_USB_CAPABILITY_DEVICE_CONNECTION_SUPER_SPEED_COMPATIBLE`

For information about the capabilities, see the Remarks section of [USBD\\_QueryUsbCapability](#).

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available.

[out, optional] OutputBuffer

A pointer to a location that receives the buffer's address. Certain capabilities may need to provide additional information to the USB device emulation class extension (UdeCx) in this buffer.

[out] ResultLength

A location that, on return, contains the size, in bytes, of the information that the callback function stored in *OutputBuffer*.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. If a capability is not supported, the client driver can return NT\_SUCCESS(status) equals FALSE, such as STATUS\_UNSUCCESSFUL.

## Remarks

The class extension invokes this callback function implemented by the client driver when the class extension receives a request to determine the emulated controller's capabilities. The callback is invoked only after [EvtDriverDeviceAdd](#) has returned, typically in [EvtDevicePrepareHardware](#). This callback cannot be invoked after [EvtDeviceReleaseHardware](#) has returned.

The class extension reports these capability GUIDs, as not supported:

- GUID\_USB\_CAPABILITY\_STATIC\_STREAMS
- GUID\_USB\_CAPABILITY\_CLEAR\_TT\_BUFFER\_ON\_ASYNC\_TRANSFER\_CANCEL

The class extension reports the GUID\_USB\_CAPABILITY\_SELECTIVE\_SUSPEND capability GUID as supported without even invoking the callback function.

For other GUIDs, the class extension invokes the client driver's implementation, such as GUID\_USB\_CAPABILITY\_CHAINED\_MDLS. The client driver is expected to determine

support for I/O requests that use a chained MDL. If this capability is supported then, the **TransferBufferMdl** member of the [URB](#) contains the request buffer. If chained MDL is not supported, the client driver does not receive **TransferBufferMdl** values that point to chained MDLs.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwdfdevice.h (include Udecx.h)
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# EVT\_UDECX\_WDF\_DEVICE\_RESET callback function (udecxwdfdevice.h)

Article 10/21/2021

The UDE client driver's implementation to reset the emulated host controller or the devices attached to it.

## Syntax

C++

```
EVT_UDECX_WDF_DEVICE_RESET EvtUdecxWdfDeviceReset;  
  
void EvtUdecxWdfDeviceReset(  
    [in] WDFDEVICE UdecxWdfDevice  
)  
{...}
```

## Parameters

[in] `UdecxWdfDevice`

A handle to a framework device object that represents the controller. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

## Return value

None

## Remarks

The USB device emulation class extension (UdeCx) invokes this callback function to notify the client driver that it must handle a reset request including resetting all downstream devices attached to the emulated host controller. This call is asynchronous. The client driver signals completion with status information by calling [UdecxWdfDeviceResetComplete](#). If the client specified [UdeWdfDeviceResetActionResetEachUsbDevice](#) in [UDECX\\_WDF\\_DEVICE\\_CONFIG](#) (during the [UdecxWdfDeviceAddUsbDeviceEmulation](#) call), this callback is never used.

Instead, each connected attached device receives an *EVT\_UDECX\_WDF\_DEVICE\_RESET* callback.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwdfdevice.h (include Udecx.h)
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UDECX\_WDF\_DEVICE\_CONFIG structure (udecxwdfdevice.h)

Article 02/22/2024

Contains pointers to event callback functions implemented by the UDE client driver for a USB host controller. Initialize this structure by calling [UDECX\\_WDF\\_DEVICE\\_CONFIG\\_INIT](#).

## Syntax

C++

```
typedef struct _UDECX_WDF_DEVICE_CONFIG {
    ULONG                                     Size;
    USHORT                                     NumberOfUsb20Ports;
    USHORT                                     NumberOfUsb30Ports;
    PFN_UDECX_WDF_DEVICE_QUERY_USB_CAPABILITY EvtUdecxWdfDeviceQueryUsbCapability;
    UDECX_WDF_DEVICE_RESET_ACTION              ResetAction;
    PFN_UDECX_WDF_DEVICE_RESET                 EvtUdecxWdfDeviceReset;
} UDECX_WDF_DEVICE_CONFIG, *PUDECX_WDF_DEVICE_CONFIG;
```

## Members

Size

The size of this structure.

NumberOfUsb20Ports

The number of USB 2.0 ports on the root hub of the emulated host controller.

NumberOfUsb30Ports

The number of USB 3.0 ports on the root hub of the emulated host controller.

EvtUdecxWdfDeviceQueryUsbCapability

A pointer to an [EVT\\_UDECX\\_WDF\\_DEVICE\\_QUERY\\_USB\\_CAPABILITY](#) callback function.

ResetAction

A [UDECX\\_WDF\\_DEVICE\\_RESET\\_ACTION](#)-type value that indicates the reset action: each attached device or the host controller.

#### EvtUdecxWdfDeviceReset

A pointer to an [EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#) callback function.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	udecxwdfdevice.h (include Udecx.h)

## See also

[UDECX\\_WDF\\_DEVICE\\_CONFIG\\_INIT](#)

[UdecxWdfDeviceAddUsbDeviceEmulation](#)

# UDECX\_WDF\_DEVICE\_CONFIG\_INIT function (udecxwdfdevice.h)

Article 02/22/2024

Initializes a [UDECX\\_WDF\\_DEVICE\\_CONFIG](#) structure.

## Syntax

C++

```
void UDECX_WDF_DEVICE_CONFIG_INIT(
    [out] PUDECX_WDF_DEVICE_CONFIG           Config,
    [in]  PFN_UDECX_WDF_DEVICE_QUERY_USB_CAPABILITY
        EvtUdecxWdfDeviceQueryUsbCapability
);
```

## Parameters

[out] Config

A pointer to a [UDECX\\_WDF\\_DEVICE\\_CONFIG](#) structure to initialize.

[in] EvtUdecxWdfDeviceQueryUsbCapability

A pointer to an [EVT\\_UDECX\\_WDF\\_DEVICE\\_QUERY\\_USB\\_CAPABILITY](#) callback function.

## Return value

None

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwddevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UDECX\\_WDF\\_DEVICE\\_CONFIG](#)

[UdecxWdfDeviceAddUsbDeviceEmulation](#)

# UDECX\_WDF\_DEVICE\_RESET\_ACTION enumeration (udecxwdfdevice.h)

Article02/22/2024

Defines values that indicate the types of reset operation supported by an emulated USB host controller.

## Syntax

C++

```
typedef enum _UDECX_WDF_DEVICE_RESET_ACTION {
    UdecxWdfDeviceResetActionResetEachUsbDevice,
    UdecxWdfDeviceResetActionResetWdfDevice
} UDECX_WDF_DEVICE_RESET_ACTION, *PUDECX_WDF_DEVICE_RESET_ACTION;
```

## Constants

[] Expand table

`UdecxWdfDeviceResetActionResetEachUsbDevice`

Each device that is attached to the controller is reset.

`UdecxWdfDeviceResetActionResetWdfDevice`

The emulated host controller is reset.

## Requirements

[] Expand table

Requirement	Value
Header	udecxwdfdevice.h (include Udecx.h)

## See also

[EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#)

## UDECX\_WDF\_DEVICE\_CONFIG

# UDECX\_WDF\_DEVICE\_RESET\_TYPE enumeration (udecxwdfdevice.h)

Article02/22/2024

Defines values that indicates the type of reset for a UDE device.

## Syntax

C++

```
typedef enum _UDECX_WDF_DEVICE_RESET_TYPE {
    UdecxWdfDeviceResetUndefined,
    UdecxWdfDeviceResetAttemptFunctionLevelDeviceReset,
    UdecxWdfDeviceResetAttemptPlatformLevelDeviceReset
} UDECX_WDF_DEVICE_RESET_TYPE, *PUDECX_WDF_DEVICE_RESET_TYPE;
```

## Constants

[ ] Expand table

<code>UdecxWdfDeviceResetUndefined</code>	Invalid type.
<code>UdecxWdfDeviceResetAttemptFunctionLevelDeviceReset</code>	Indicates a function level reset.
<code>UdecxWdfDeviceResetAttemptPlatformLevelDeviceReset</code>	Indicates a platform level reset.

## Requirements

[ ] Expand table

Requirement	Value
Header	udecxwdfdevice.h

# UdecxInitializeWdfDeviceInit function (udecxwdfdevice.h)

Article02/22/2024

Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

## Syntax

C++

```
NTSTATUS UdecxInitializeWdfDeviceInit(  
    PWDFDEVICE_INIT WdfDeviceInit  
) ;
```

## Parameters

`WdfDeviceInit`

A pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the emulated host controller device calls this method in its [EvtDriverDeviceAdd](#) implementation before it calls [WdfDeviceCreate](#) and [UdecxWdfDeviceAddUsbDeviceEmulation](#). For code example, see [UdecxWdfDeviceAddUsbDeviceEmulation](#).

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwdfdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[WDFDEVICE\\_INIT](#)

[WdfDeviceCreate](#)

[Write a UDE client driver](#)

# UdecxWdfDeviceAddUsbDeviceEmulation function (udecxwdfdevice.h)

Article 02/22/2024

Initializes a framework device object to support operations related to a host controller and a virtual USB device attached to the controller.

## Syntax

C++

```
NTSTATUS UdecxWdfDeviceAddUsbDeviceEmulation(
    WDFDEVICE             WdfDevice,
    [in] PUDECX_WDF_DEVICE_CONFIG Config
);
```

## Parameters

`WdfDevice`

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

`[in] Config`

A pointer to a [UDECX\\_WDF\\_DEVICE\\_CONFIG](#) structure that the client driver initialized by calling [UDECX\\_WDF\\_DEVICE\\_CONFIG\\_INIT](#).

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

The UDE client driver for the emulated host controller and the USB device must call this method after the [WdfDeviceCreate](#) call.

During this call, the client driver-supplied event callback implementations are also registered. Supply function pointers to those functions by call setting appropriate members of [UDECX\\_WDF\\_DEVICE\\_CONFIG](#).

The method makes the framework device object capable of performing operations related to a controller and its root hub, such as handling various queues required to process IOCTL requests sent to the attached USB device.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwdfdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# UdecxWdfDeviceNeedsReset function (udecxwdfdevice.h)

Article 05/23/2024

Informs the USB device emulation class extension (UdeCx) that the device needs a reset operation.

## Syntax

C++

```
NTSTATUS UdecxWdfDeviceNeedsReset(
    [In] WDFDEVICE                 UdeWdfDevice,
    [In] UDECX_WDF_DEVICE_RESET_TYPE ResetType
);
```

## Parameters

[In] `UdeWdfDevice`

A handle to a framework device object that represents a USB device. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

[In] `ResetType`

A [UDECX\\_WDF\\_DEVICE\\_RESET\\_TYPE](#)-type value that indicates the type of reset. Only `UdecxWdfDeviceResetAttemptPlatformLevelDeviceReset` is supported at this time.

## Return value

The function returns `STATUS_SUCCESS` if the operation succeeds. Otherwise, returns an appropriate [NTSTATUS](#) error code.

## Remarks

If an existing reset operation is in progress, the function fails with a `STATUS_DEVICE_BUSY` error. Note that only `UdecxWdfDeviceResetAttemptPlatformLevelDeviceReset` is supported at this time. `UdecxWdfDeviceResetAttemptFunctionLevelDeviceReset` is not supported.

# Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	udecxwdfdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[UDECX\\_WDF\\_DEVICE\\_RESET\\_TYPE](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

# UdecxWdfDeviceResetComplete function (udecxwdfdevice.h)

Article04/01/2021

Informs the USB device emulation class extension (UdeCx) that the reset operation on the specified controller has completed.

## Syntax

C++

```
void UdecxWdfDeviceResetComplete(
    WDFDEVICE UdeWdfDevice
);
```

## Parameters

UdeWdfDevice

A handle to a framework device object that represents the controller that has been reset. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

## Return value

None

## Remarks

When the class extension calls the [EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#) callback function, that call is asynchronous. The client driver must call [UdecxWdfDeviceResetComplete](#) to notify the class extension when the reset operation is complete with appropriate status information.

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwdfdevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	<=DISPATCH_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[EVT\\_UDECX\\_WDF\\_DEVICE\\_RESET](#)

[Write a UDE client driver](#)

# UdecxWdfDeviceTryHandleUserIoctl function (udecxwdfdevice.h)

Article 02/22/2024

Attempts to handle an IOCTL request sent by a user-mode software.

## Syntax

C++

```
BOOLEAN UdecxWdfDeviceTryHandleUserIoctl(
    WDFDEVICE UdecxWdfDevice,
    [in] WDFREQUEST Request
);
```

## Parameters

`UdecxWdfDevice`

A handle to a framework device object that represents the controller. The client driver initialized this object in the previous call to [UdecxWdfDeviceAddUsbDeviceEmulation](#).

`[in] Request`

A handle to a framework request object that represents the IOCTL request.

## Return value

TRUE indicates that USB device emulation class extension (UdeCx) recognized and completed the request (with success or failure). In this case, the client driver must not complete the request. FALSE otherwise; the driver must complete the request.

## Remarks

The UDE client driver presents itself to user-mode software as a host controller driver. The client driver registers and exposes the GUID\_DEVINTERFACE\_USB\_HOST\_CONTROLLER device interface GUID. User-mode software can open a handle to the device by specifying that GUID. By using that handle, the software can send IOCTL requests.

**Note** Note that other interface's IOCTL codes may overlap with the USB host controller interface. If such I/O reaches this function the IOCTL will not be handled correctly.

The client driver does not need to process the received IOCTL. It can send the request to the class extension by calling `UdecxWdfDeviceTryHandleUserIoctl`. If the class extension recognizes the request as a standard request, it completes it. Otherwise, the call fails and the client driver is then expected to complete the request. For a list of IOCTLs that must be handled, see [USB IOCTLs for applications and services](#).

- [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#)
- [IOCTL\\_USB\\_GET\\_ROOT\\_HUB\\_NAME](#)
- [IOCTL\\_USB\\_USER\\_REQUEST](#)

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	udecxwddevice.h (include Udecx.h)
Library	Udecxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[Architecture: USB Device Emulation \(UDE\)](#)

[Write a UDE client driver](#)

# ufxbase.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ufxbase.h contains the following programming interfaces:

## IOCTLs

### [IOCTL\\_INTERNAL\\_USBFN\\_DESCRIPTOR\\_UPDATE](#)

The USB function class extension sends this request to the client driver to update to the endpoint descriptor for the specified endpoint.

## Structures

### [UFX\\_DEVICE\\_CAPABILITIES](#)

The UFX\_DEVICE\_CAPABILITIES structure is used USB to define properties of the Universal Serial Bus (USB) device created by the controller.

### [UFX\\_HARDWARE\\_FAILURE\\_CONTEXT](#)

The UFX\_HARDWARE\_FAILURE\_CONTEXT structure is used to define controller-specific hardware failure properties.

## Enumerations

### [UFX\\_CLASS\\_FUNCTIONS](#)

Learn more about: [\\_UFX\\_CLASS\\_FUNCTIONS](#) enumeration

### [USBFN\\_ACTION](#)

Defines special actions UFX should take when the client driver calls the `UfxDevicePortDetectCompleteEx` function.

# IOCTL\_INTERNAL\_USBFN\_DESCRIPTOR\_UPDATE IOCTL (ufxbase.h)

Article06/16/2023

The USB function class extension sends this request to the client driver to update to the endpoint descriptor for the specified endpoint.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

The input buffer points to a [USBFNPIPEID](#) that specifies the pipe ID for the endpoint.

## Input buffer length

The size of a [USBFNPIPEID](#) value.

## Output buffer

The output buffer points to a [USB\\_ENDPOINT\\_DESCRIPTOR](#) structure that describes the endpoint descriptor. To retrieve the structure, the client driver must call [WdfRequestRetrieveOutputBuffer](#).

## Output buffer length

The size of a [USB\\_ENDPOINT\\_DESCRIPTOR](#) structure.

## Status block

The client driver shall complete the request with [STATUS\\_SUCCESS](#) if the request is successful. Otherwise, the client driver shall complete the driver with to the appropriate error condition, such as [STATUS\\_INVALID\\_PARAMETER](#) or [STATUS\\_INSUFFICIENT\\_RESOURCES](#).

## Remarks

UFX sends this IOCTL to the command queue created for the endpoint by [UfxEndpointCreate](#). The client driver is expected to update the configuration of the endpoint on the controller with the parameters contained in the endpoint descriptor.

## Requirements

Header	ufxbase.h
--------	-----------

# UFX\_CLASS\_FUNCTIONS enumeration (ufxbase.h)

Article 07/27/2021

Defines index values for all export functions for the UFX class extension.

## Syntax

C++

```
typedef enum _UFX_CLASS_FUNCTIONS {
    UfxFdoInitIndex = ,
    UfxDeviceCreateIndex = ,
    UfxDeviceEventCompleteIndex = ,
    UfxDeviceNotifyHardwareReadyIndex = ,
    UfxDeviceNotifyAttachIndex = ,
    UfxDeviceNotifyDetachIndex = ,
    UfxDeviceNotifySuspendIndex = ,
    UfxDeviceNotifyResumeIndex = ,
    UfxDeviceNotifyResetIndex = ,
    UfxDevicePortDetectCompleteIndex = ,
    UfxDeviceProprietaryChargerDetectCompleteIndex = ,
    UfxDeviceNotifyHardwareFailureIndex = ,
    UfxDeviceIoControlIndex = ,
    UfxDeviceIoInternalControlIndex = ,
    UfxEndpointCreateIndex = ,
    UfxEndpointInitSetEventCallbacksIndex = ,
    UfxEndpointNotifySetupIndex = ,
    UfxEndpointGetTransferQueueIndex = ,
    UfxEndpointGetCommandQueueIndex = ,
    UfxDevicePortDetectCompleteExIndex = ,
    UfxDeviceNotifyFinalExitIndex = ,
    UfxFunctionsMax =
} UFX_CLASS_FUNCTIONS;
```

## Constants

**UfxFdoInitIndex**

Index for [UfxFdoinit](#).

**UfxDeviceCreateIndex**

Index for [UfxDeviceCreate](#).

`UfxDeviceEventCompleteIndex`

Index for [UfxDeviceEventComplete](#).

`UfxDeviceNotifyHardwareReadyIndex`

Index for [UfxDeviceNotifyHardwareReady](#).

`UfxDeviceNotifyAttachIndex`

Index for [UfxDeviceNotifyAttach](#).

`UfxDeviceNotifyDetachIndex`

Index for [UfxDeviceNotifyDetach](#).

`UfxDeviceNotifySuspendIndex`

Index for [UfxDeviceNotifySuspend](#).

`UfxDeviceNotifyResumeIndex`

Index for [UfxDeviceNotifyResume](#).

`UfxDeviceNotifyResetIndex`

Index for [UfxDeviceNotifyResetIndex](#).

`UfxDevicePortDetectCompleteIndex`

Index for [UfxDevicePortDetectComplete](#).

`UfxDeviceProprietaryChargerDetectCompleteIndex`

Index for [UfxDeviceProprietaryChargerDetectComplete](#).

`UfxDeviceNotifyHardwareFailureIndex`

Index for [UfxDeviceNotifyHardwareFailure](#).

`UfxDeviceIoControlIndex`

Index for [UfxDeviceIoControl](#).

`UfxDeviceIoInternalControlIndex`

Index for [UfxDeviceIoInternalControl](#).

`UfxEndpointCreateIndex`

Index for [UfxEndpointCreate](#).

`UfxEndpointInitSetEventCallbacksIndex`

Index for [UfxEndpointInitSetEventCallbacks](#).

`UfxEndpointNotifySetupIndex`

Index for [UfxEndpointNotifySetup](#).

`UfxEndpointGetTransferQueueIndex`

Index for [UfxEndpointGetTransferQueue](#)

`UfxEndpointGetCommandQueueIndex`

Index for [UfxEndpointGetCommandQueue](#).

`UfxDevicePortDetectCompleteExIndex`

Index for [UfxDevicePortDetectCompleteEx](#).

`UfxDeviceNotifyFinalExitIndex`

Index for [UfxDeviceNotifyFinalExit](#).

`UfxFunctionsMax`

## Requirements

Minimum supported client	Windows
Header	ufxbase.h

# UFX\_DEVICE\_CAPABILITIES structure (ufxbase.h)

Article 02/22/2024

The **UFX\_DEVICE\_CAPABILITIES** structure is used USB to define properties of the Universal Serial Bus (USB) device created by the controller.

## Syntax

C++

```
typedef struct _UFX_DEVICE_CAPABILITIES {
    ULONG             Size;
    USB_DEVICE_SPEED MaxSpeed;
    ULONG             RemoteWakeSignalDelay;
    BOOLEAN           PdcpSupported;
    USHORT            InEndpointBitmap;
    USHORT            OutEndpointBitmap;
    BOOLEAN           SharesConnectors;
    ULONG             GroupId;
} UFX_DEVICE_CAPABILITIES, *PUFX_DEVICE_CAPABILITIES;
```

## Members

Size

Size of the **UFX\_DEVICE\_CAPABILITIES** structure.

MaxSpeed

Indicates the maximum USB speed supported by the device.

RemoteWakeSignalDelay

The minimum time interval in milliseconds to wait after being suspended before requesting remote wakeup.

PdcpSupported

If **true**, indicates the client driver supports proprietary charger detection.

InEndpointBitmap

A bitmap that defines which endpoint numbers can support an IN endpoint. Bit 0 indicates endpoint address 0, bit 1 indicates endpoint address 1, etc. Bit 0 (the default control endpoint) is required to be set to 1.

#### OutEndpointBitmap

A bitmap that defines which endpoint numbers can support an OUT endpoint. Bit 0 indicates endpoint address 0, bit 1 indicates endpoint address 1, etc. Bit 0 (the default control endpoint) is required to be set to 1.

#### SharesConnectors

A boolean value that indicates the connector supports multiple connectors.

#### GroupId

The group identifier the shared connector group to which a controller will be added.

## Requirements

[Expand table](#)

Requirement	Value
Header	ufxbase.h

# UFX\_HARDWARE\_FAILURE\_CONTEXT structure (ufxbase.h)

Article02/22/2024

The **UFX\_HARDWARE\_FAILURE\_CONTEXT** structure is used to define controller-specific hardware failure properties.

## Syntax

C++

```
typedef struct _UFX_HARDWARE_FAILURE_CONTEXT {
    ULONG Size;
    ULONG ExceptionCode;
    UCHAR Data[1];
} UFX_HARDWARE_FAILURE_CONTEXT, *PUFX_HARDWARE_FAILURE_CONTEXT;
```

## Members

Size

The size of the **UFX\_HARDWARE\_FAILURE\_CONTEXT** structure.

ExceptionCode

The controller-specific hardware failure code.

Data[1]

A variable-length array of data associated with the hardware failure.

## Remarks

In cases where the function controller has experienced a fatal error, the client driver may allocate a variable-length **UFX\_HARDWARE\_FAILURE\_CONTEXT** structure, set the **Size** field to the allocated size, set the **ExceptionCode** field to a value indicating the type of hardware error (as defined by the client driver) and fill in any associated information in the **Data** array. It may then pass this structure to the [UfxDeviceNotifyHardwareFailure](#) UFX function. UFX will in turn pass this structure to the client driver's [EVT\\_UFX\\_DEVICE\\_CONTROLLER\\_RESET](#) event callback function (if it exists).

# Requirements

 Expand table

Requirement	Value
Header	ufxbase.h

# USBFN\_ACTION enumeration (ufxbase.h)

Article 02/22/2024

Defines special actions UFX should take when the client driver calls the [UfxDevicePortDetectCompleteEx](#) function.

## Syntax

C++

```
typedef enum _USBFN_ACTION {
    UsbfnActionNone,
    UsbfnActionNoCad,
    UsbfnActionDetectProprietaryCharger
} USBFN_ACTION, *PUSBFN_ACTION;
```

## Constants

[\[\] Expand table](#)

<code>UsbfnActionNone</code>	No special action should be taken.
<code>UsbfnActionNoCad</code>	UFX should not notify the battery manager about the detected port type or the maximum current that can be drawn from the upstream port.
<code>UsbfnActionDetectProprietaryCharger</code>	UFX should initiate proprietary charger detection by calling the client driver's <a href="#">EVT_UFX_DEVICE_DETECT_PROPRIETARY_CHARGER</a> callback function.

## Requirements

[\[\] Expand table](#)

Requirement	Value
Header	ufxbase.h

# ufxclient.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ufxclient.h contains the following programming interfaces:

## Functions

### [UFX\\_DEVICE\\_CALLBACKS\\_INIT](#)

The UFX\_DEVICE\_CALLBACKS\_INIT macro initializes the UFX\_DEVICE\_CALLBACKS structure.

### [UFX\\_DEVICE\\_CAPABILITIES\\_INIT](#)

The UFX\_DEVICE\_CAPABILITIES\_INIT macro initializes the UFX\_DEVICE\_CAPABILITIES structure.

### [UFX\\_ENDPOINT\\_CALLBACKS\\_INIT](#)

The UFX\_ENDPOINT\_CALLBACKS\_INIT macro initializes the UFX\_ENDPOINT\_CALLBACKS structure.

### [UfxDeviceCreate](#)

Creates a UFX device object, registers event callback routines, and specifies capabilities specific to the controller.

### [UfxDeviceEventComplete](#)

Informs UFX that the client driver has completed processing a UFX callback function.

### [UfxDeviceIoControl](#)

Passes non-internal IOCTLs from user-mode to UFX.

### [UfxDeviceIoInternalControl](#)

Passes kernel mode IOCTLs to UFX.

### [UfxDeviceNotifyAttach](#)

Notifies UFX that the device's USB cable has been attached.

### [UfxDeviceNotifyDetach](#)

	Notifies UFX that the device's USB cable has been detached.
<a href="#">UfxDeviceNotifyFinalExit</a>	Notifies UFX that the device is detached.
<a href="#">UfxDeviceNotifyHardwareFailure</a>	Notifies UFX about a non-recoverable hardware failure in the controller.
<a href="#">UfxDeviceNotifyHardwareReady</a>	Notifies UFX that the hardware is ready.
<a href="#">UfxDeviceNotifyReset</a>	Notifies UFX about a USB bus reset event.
<a href="#">UfxDeviceNotifyResume</a>	Notifies UFX about a USB bus resume event.
<a href="#">UfxDeviceNotifySuspend</a>	Notifies UFX about a USB bus suspend event.
<a href="#">UfxDevicePortDetectComplete</a>	Notifies UFX about the port type that was detected.
<a href="#">UfxDevicePortDetectCompleteEx</a>	Notifies UFX about the port type that was detected, and optionally requests an action.
<a href="#">UfxDeviceProprietaryChargerDetectComplete</a>	Notifies UFX about a detected proprietary port/charger type.
<a href="#">UfxEndpointCreate</a>	Creates an endpoint object.
<a href="#">UfxEndpointGetCommandQueue</a>	Returns the command queue previously created by UfxEndpointCreate.
<a href="#">UfxEndpointGetTransferQueue</a>	Returns the transfer queue previously created by UfxEndpointCreate.

<a href="#">UfxEndpointInitSetEventCallbacks</a>	Initialize a UFXENDPOINT_INIT structure.
<a href="#">UfxEndpointNotifySetup</a>	Notifies UFX when the client driver receives a setup packet from the host.
<a href="#">UfxFdInit</a>	Initializes the WDFDEVICE_INIT structure that the client driver subsequently provides when it calls <code>WdfDeviceCreate</code> .

## Callback functions

<a href="#">EVT_UFX_DEVICE_ADDRESSED</a>	The client driver's implementation to assign an address on the function controller.
<a href="#">EVT_UFX_DEVICE_CONTROLLER_RESET</a>	The client driver's implementation to reset the function controller to its initial state.
<a href="#">EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD</a>	The client driver's implementation to create a default control endpoint.
<a href="#">EVT_UFX_DEVICE_ENDPOINT_ADD</a>	The client driver's implementation to create a default endpoint object.
<a href="#">EVT_UFX_DEVICE_HOST_CONNECT</a>	The client driver's implementation to initiate connection with the host.
<a href="#">EVT_UFX_DEVICE_HOST_DISCONNECT</a>	The client driver's implementation to disable the function controller's communication with the host.
<a href="#">EVT_UFX_DEVICE_PORT_CHANGE</a>	The client driver's implementation to update the type of the new port to which the USB device is connected.

#### [EVT\\_UFX\\_DEVICE\\_PORT\\_DETECT](#)

The client driver's implementation to initiate port detection.

#### [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_DETECT](#)

The client driver's implementation to initiate proprietary charger detection.

#### [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_RESET](#)

The client driver's implementation to resets proprietary charger.

#### [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_SET\\_PROPERTY](#)

The client driver's implementation to set charger information that it uses to enable charging over USB.

#### [EVT\\_UFX\\_DEVICE\\_REMOTE\\_WAKEUP\\_SIGNAL](#)

The client driver's implementation to initiate remote wake-up on the function controller.

#### [EVT\\_UFX\\_DEVICE\\_SUPER\\_SPEED\\_POWER\\_FEATURE](#)

The client driver's implementation to set or clear the specified power feature on the function controller.

#### [EVT\\_UFX\\_DEVICE\\_TEST\\_MODE\\_SET](#)

The client driver's implementation to set the test mode of the function controller.

#### [EVT\\_UFX\\_DEVICE\\_TESTHOOK](#)

This IOCTL code is not supported.

#### [EVT\\_UFX\\_DEVICE\\_USB\\_STATE\\_CHANGE](#)

The client driver's implementation to update the state of the USB device.

## Structures

#### [UFX\\_DEVICE\\_CALLBACKS](#)

The UFX\_DEVICE\_CALLBACKS structure is used to define then event callback functions supported by the client driver.

## [UFX\\_ENDPOINT\\_CALLBACKS](#)

The UFX\_ENDPOINT\_CALLBACKS structure is used to define the event callback functions supported by the client driver.

# EVT\_UFX\_DEVICE\_ADDRESSED callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to assign an address on the function controller.

## Syntax

C++

```
EVT_UFX_DEVICE_ADDRESSED EvtUfxDeviceAddressed;

void EvtUfxDeviceAddressed(
    [in] UFXDEVICE unnamedParam1,
    [in] USHORT unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

New USB device address to assign.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_ADDRESSED* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_ADDRESSED UfxDevice_EvtDeviceAddressed;

VOID
UfxDevice_EvtDeviceAddressed (
    _In_ UFXDEVICE UfxDevice,
    _In_ USHORT DeviceAddress
)
/*++

Routine Description:

    EvtDeviceAddressed handler for the UFXDEVICE object.
    Sets the Address indicated by 'DeviceAddress' on the controller.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    DeviceAddress - USB Device Address, as determined by the UFX.

--*/
{
    UNREFERENCED_PARAMETER(DeviceAddress);

    TraceEntry();

    //
    // Set the device address on the controller
    //

    //
    // ##### Insert code to set the device address on controller #####
    //

    UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);

    TraceExit();
}
```

## Requirements

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	<=DISPATCH_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_CONTROLLER\_RESET callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to reset the function controller to its initial state.

## Syntax

C++

```
EVT_UFX_DEVICE_CONTROLLER_RESET EvtUfxDeviceControllerReset;

void EvtUfxDeviceControllerReset(
    [in]           UFXDEVICE unnamedParam1,
    [in, optional] PUFX_HARDWARE_FAILURE_CONTEXT unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in, optional] unnamedParam2

A pointer to a variable-length UFX\_HARDWARE\_FAILURE\_CONTEXT structure allocated by the client driver.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_CONTROLLER\_RESET* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Requirements

 [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_DEFAULT\_ENDPOINT\_ADD callback function (ufxclient.h)

Article 02/22/2024

The client driver's implementation to create a default control endpoint.

## Syntax

C++

```
EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD EvtUfxDeviceDefaultEndpointAdd;

void EvtUfxDeviceDefaultEndpointAdd(
    [in]      UFXDEVICE unnamedParam1,
    [in]      USHORT unnamedParam2,
    [in, out] PUFXENDPOINT_INIT unnamedParam3
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

The default maximum packet size that can be sent from or to this endpoint.

[in, out] unnamedParam3

A pointer to an UFXENDPOINT\_INIT opaque structure that contains the endpoint descriptor required to create an endpoint object.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_DEFAULT\_ENDPOINT\_ADD* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

To create the endpoint the client driver is expected to initialize the attributes of the endpoint's transfer and command queues, and then call [UfxEndpointCreate](#) to create the endpoint. After the default control endpoint is created, UFX is ready to process setup packets and other control transfer packets from host.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_DEFAULT_ENDPOINT_ADD UfxDevice_EvtDeviceDefaultEndpointAdd;

VOID
UfxDevice_EvtDeviceDefaultEndpointAdd (
    _In_ UFXDEVICE UfxDevice,
    _In_ USHORT MaxPacketSize,
    _Inout_ PUFXENDPOINT_INIT EndpointInit
)
/*++

Routine Description:

    EvtDeviceDefaultEndpointAdd handler for the UFXDEVICE object.
    Creates UFXENDPOINT object corresponding to the default endpoint of the
    device.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    MaxPacketSize - Max packet size of the device's default endpoint.

    EndpointInit - Pointer to the Opaque UFXENDPOINT_INIT object

--*/
{
    NTSTATUS Status;
    USB_ENDPOINT_DESCRIPTOR Descriptor;

    PAGED_CODE();

    TraceEntry();
}
```

```

Descriptor.bDescriptorType = USB_ENDPOINT_DESCRIPTOR_TYPE;
Descriptor.bEndpointAddress = 0;
Descriptor.bInterval = 0;
Descriptor.bLength = sizeof(USB_ENDPOINT_DESCRIPTOR);
Descriptor.bmAttributes = USB_ENDPOINT_TYPE_CONTROL;
Descriptor.wMaxPacketSize = MaxPacketSize;

// ##### TODO: Insert code to add the endpoint.
// See code example for EVT_UFX_DEVICE_ENDPOINT_ADD #####
}

End:
UfxDeviceEventComplete(UfxDevice, Status);
TraceExit();
}

```

## Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_ENDPOINT\_ADD callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to create a default endpoint object.

## Syntax

C++

```
EVT_UFX_DEVICE_ENDPOINT_ADD EvtUfxDeviceEndpointAdd;

NTSTATUS EvtUfxDeviceEndpointAdd(
    [in]      UFXDEVICE unnamedParam1,
    [in]      const PUSB_ENDPOINT_DESCRIPTOR unnamedParam2,
    [in, out] PUFXENDPOINT_INIT unnamedParam3
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

A pointer to a [USB\\_ENDPOINT\\_DESCRIPTOR](#) structure that contains descriptor data.

[in, out] unnamedParam3

A pointer to an UFXENDPOINT\_INIT opaque structure that contains the endpoint descriptor required to create an endpoint object.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

# Remarks

The client driver for the function host controller registers its `EVT_UFX_DEVICE_ENDPOINT_ADD` implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

To create the endpoint the client driver is expected to initialize the attributes of the endpoint's transfer and command queues, and then call [UfxEndpointCreate](#) to create the endpoint.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_ENDPOINT_ADD UfxDevice_EvtDeviceEndpointAdd;

NTSTATUS
UfxDevice_EvtDeviceEndpointAdd (
    _In_ UFXDEVICE UfxDevice,
    _In_ const PUSB_ENDPOINT_DESCRIPTOR EndpointDescriptor,
    _Inout_ PUFXENDPOINT_INIT EndpointInit
)
/*++

Routine Description:

    EvtDeviceEndpointAdd handler for the UFXDEVICE object.
    Creates UFXENDPOINT object corresponding to the newly reported endpoint.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    EndpointDescriptor - Constant Pointer to Endpoint descriptor for the
        newly reported endpoint.

    EndpointInit - Pointer to the Opaque UFXENDPOINT_INIT object

Return Value:

    STATUS_SUCCESS on success, or an appropriate NTSTATUS message on
    failure.

--*/
{
    NTSTATUS Status;
    WDF_OBJECT_ATTRIBUTES Attributes;
```

```

WDF_IO_QUEUE_CONFIG TransferQueueConfig;
WDF_OBJECT_ATTRIBUTES TransferQueueAttributes;
WDF_IO_QUEUE_CONFIG CommandQueueConfig;
WDF_OBJECT_ATTRIBUTES CommandQueueAttributes;
UFXENDPOINT Endpoint;
PUFXENDPOINT_CONTEXT EpContext;
PUFXDEVICE_CONTEXT DeviceContext;
UFX_ENDPOINT_CALLBACKS Callbacks;
PENDPOINT_QUEUE_CONTEXT QueueContext;
WDFQUEUE Queue;

TraceEntry();

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&Attributes,
UFXENDPOINT_CONTEXT);
Attributes.ExecutionLevel = WdfExecutionLevelPassive;
Attributes.EvtCleanupCallback = UfxEndpoint_Cleanup;

//
// Note: Execution level needs to be passive to avoid deadlocks with
WdfRequestComplete.

//
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&TransferQueueAttributes,
ENDPOINT_QUEUE_CONTEXT);
TransferQueueAttributes.ExecutionLevel = WdfExecutionLevelPassive;

WDF_IO_QUEUE_CONFIG_INIT(&TransferQueueConfig,
WdfIoQueueDispatchManual);
TransferQueueConfig.AllowZeroLengthRequests = TRUE;
TransferQueueConfig.EvtIoStop = EndpointQueue_EvtIoStop;

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&CommandQueueAttributes,
ENDPOINT_QUEUE_CONTEXT);
CommandQueueAttributes.ExecutionLevel = WdfExecutionLevelPassive;

WDF_IO_QUEUE_CONFIG_INIT(&CommandQueueConfig,
WdfIoQueueDispatchSequential);
CommandQueueConfig.EvtIoInternalDeviceControl = EvtEndpointCommandQueue;

UFX_ENDPOINT_CALLBACKS_INIT(&Callbacks);
UfxEndpointInitSetEventCallbacks(EndpointInit, &Callbacks);

Status = UfxEndpointCreate(
    Device,
    EndpointInit,
    &Attributes,
    &TransferQueueConfig,
    &TransferQueueAttributes,
    &CommandQueueConfig,
    &CommandQueueAttributes,
    &Endpoint);

Status = WdfCollectionAdd(DeviceContext->Endpoints, Endpoint);

```

```

EpContext = UfxEndpointGetContext(Endpoint);
EpContext->UfxDevice = Device;
EpContext->WdfDevice = DeviceContext->FdowdfDevice;
RtlCopyMemory(&EpContext->Descriptor, Descriptor, sizeof(*Descriptor));

Queue = UfxEndpointGetTransferQueue(Endpoint);
QueueContext = EndpointQueueGetContext(Queue);
QueueContext->Endpoint = Endpoint;

Queue = UfxEndpointGetCommandQueue(Endpoint);
QueueContext = EndpointQueueGetContext(Queue);
QueueContext->Endpoint = Endpoint;

//  

// This can happen if we're handling a SetInterface command.  

//  

if (DeviceContext->UsbState == UsbfnDeviceStateConfigured) {  

    UfxEndpointConfigure(Endpoint);  

}

Status = WdfIoQueueReadyNotify(  

    UfxEndpointGetTransferQueue(Endpoint),  

    TransferReadyNotify,  

    Endpoint);  

  

End:  

    TraceExit();  

    return Status;  

}

```

## Requirements

[\[\] Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_HOST\_CONNECT callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to initiate connection with the host.

## Syntax

C++

```
EVT_UFX_DEVICE_HOST_CONNECT EvtUfxDeviceHostConnect;

void EvtUfxDeviceHostConnect(
    [in] UFXDEVICE unnamedParam1
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_HOST\_CONNECT* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver must not initiate connection with the host until UFX invokes this event callback. The client driver shall indicate completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

C++

```
EVT_UFX_DEVICE_HOST_CONNECT UfxDevice_EvtDeviceHostConnect;

VOID
UfxDevice_EvtDeviceHostConnect (
    _In_ UFXDEVICE UfxDevice
)
/*++

Routine Description:

    EvtDeviceHostConnect callback handler for UFXDEVICE object.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

--*/
{
    PCONTROLLER_CONTEXT ControllerContext;
    PUFXDEVICE_CONTEXT DeviceContext;
    BOOLEAN EventComplete;

    TraceEntry();

    DeviceContext = UfxDeviceGetContext(UfxDevice);
    ControllerContext = DeviceGetControllerContext(DeviceContext-
>FdowdfDevice);

    EventComplete = TRUE;

    WdfSpinLockAcquire(ControllerContext->DpcLock);

    //
    // ##### TODO: Insert code to set the run state on the controller #####
    //

    WdfSpinLockRelease(ControllerContext->DpcLock);

    if (EventComplete) {
        UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);
    }

    TraceExit();
}
```

## Requirements

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	<=DISPATCH_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_HOST\_DISCONNECT callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to disable the function controller's communication with the host.

## Syntax

C++

```
EVT_UFX_DEVICE_HOST_DISCONNECT EvtUfxDeviceHostDisconnect;  
  
void EvtUfxDeviceHostDisconnect(  
    [in] UFXDEVICE unnamedParam1  
)  
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_HOST\_DISCONNECT* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

UFX invokes this event callback to perform a soft-disconnect on the USB cable. After this call, the client driver must not initiate a connection with the host until UFX invokes [EVT\\_UFX\\_DEVICE\\_HOST\\_CONNECT](#).

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

C++

```
EVT_UFX_DEVICE_HOST_DISCONNECT UfxDevice_EvtDeviceHostDisconnect;

VOID
UfxDevice_EvtDeviceHostDisconnect (
    _In_ UFXDEVICE UfxDevice
)
/*++

Routine Description:

    EvtDeviceHostDisconnect callback handler for UFXDEVICE object.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

--*/
{
    PCONTROLLER_CONTEXT ControllerContext;
    PUFXDEVICE_CONTEXT DeviceContext;
    BOOLEAN EventComplete;

    TraceEntry();

    DeviceContext = UfxDeviceGetContext(UfxDevice);
    ControllerContext = DeviceGetControllerContext(DeviceContext-
>FdoWdfDevice);

    EventComplete = TRUE;

    //
    // ##### TODO: Cancel all transfers. #####
    //

    WdfSpinLockAcquire(ControllerContext->DpcLock);

    //
    // ##### TODO: Insert code to clear the run state on the controller #####
    //

    WdfSpinLockRelease(ControllerContext->DpcLock);

    if (EventComplete) {
        UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);
    }
}
```

```
    TraceExit();  
}
```

# Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	<=DISPATCH_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_PORT\_CHANGE callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to update the type of the new port to which the USB device is connected.

## Syntax

C++

```
EVT_UFX_DEVICE_PORT_CHANGE EvtUfxDevicePortChange;

void EvtUfxDevicePortChange(
    [in] UFXDEVICE unnamedParam1,
    [in] USBFN_PORT_TYPE unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

A USBFN\_PORT\_STATE-typed flag that indicates the type of the new port.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_PORT\_CHANGE* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

UFX invokes this event callback to inform the client driver about the new state of the device.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_PORT_CHANGE UfxDevice_EvtDevicePortChange;

VOID
UfxDevice_EvtDevicePortChange (
    _In_          UFXDEVICE UfxDevice,
    _In_          USBFN_PORT_TYPE NewPort
)
/*++

Routine Description:

    EvtDevicePortChange handler for the UFXDEVICE object.
    Caches the new port type, and stops or resumes idle as needed.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    NewPort - New port type

--*/
{
    NTSTATUS Status;
    PUFXDEVICE_CONTEXT Context;

    PAGED_CODE();

    TraceEntry();

    Context = UfxDeviceGetContext(UfxDevice);

    TraceInformation("New PORT: %d", NewPort);

    //
    // ##### TODO: Insert code to examine the device USB state and port type
    //           and determine if it needs to stop or resume idle.

    UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);
```

```
    TraceExit();  
}
```

# Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_PORT\_DETECT callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to initiate port detection

## Syntax

C++

```
EVT_UFX_DEVICE_PORT_DETECT EvtUfxDevicePortDetect;  
  
void EvtUfxDevicePortDetect(  
    [in] UFXDEVICE unnamedParam1  
)  
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_PORT\_DETECT* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver must indicate completion of port detection by calling the [UfxDevicePortDetectComplete](#) or [UfxDevicePortDetectCompleteEx](#) methods.

## Examples

C++

```
EVT_UFX_DEVICE_PORT_DETECT UfxDevice_EvtDevicePortDetect;

VOID
UfxDevice_EvtDevicePortDetect (
    _In_ UFXDEVICE UfxDevice
)
/*++
Routine Description:

    Starts the port detection state machine

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

--*/
{
    PUFXDEVICE_CONTEXT DeviceContext;
    PCONTROLLER_CONTEXT ControllerContext;

    DeviceContext = UfxDeviceGetContext(UfxDevice);
    ControllerContext = DeviceGetControllerContext(DeviceContext-
>FdoWdfDevice);

    //
    // ##### TODO: Insert code to determine port/charger type #####
    //
    // In this example we will return an unknown port type.
    // This will allow UFX to connect to a host if one is present.
    // UFX will timeout after 5 seconds if no host is present and transition
    to
    // an invalid charger type, which will allow the controller to exit D0.
    //

    UfxDevicePortDetectComplete(ControllerContext->UfxDevice,
UsbfnUnknownPort);
}
```

## Requirements

[\[\] Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0

Requirement	Value
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceEventComplete](#)
- [UfxDevicePortDetectComplete](#)
- [UfxDevicePortDetectCompleteEx](#)

# EVT\_UFX\_DEVICE\_PROPRIETARY\_CHARGER\_DETECT callback function (ufxclient.h)

Article01/20/2022

The client driver's implementation to initiate proprietary charger detection.

## Syntax

C++

```
EVT_UFX_DEVICE_PROPRIETARY_CHARGER_DETECT
EvtUfxDeviceProprietaryChargerDetect;

void EvtUfxDeviceProprietaryChargerDetect(
    [in] UFXDEVICE unnamedParam1
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

*EVT\_UFX\_DEVICE\_DETECT\_PROPRIETARY\_CHARGER* is an optional event callback. The client driver is required to implement this event callback only if it supports proprietary charger detection. The driver indicates support in the [UfxDeviceCreate](#) call by setting **PdcpSupported** to TRUE in [UFX\\_DEVICE\\_CAPABILITIES](#). If the client driver does not support the functionality, the [EvtDeviceProprietaryChargerDetect](#), [EvtDeviceProprietaryChargerSetProperty](#), and [EvtDeviceProprietaryChargerReset](#)

members of the [UFX\\_DEVICE\\_CALLBACKS](#) structure must be set to NULL in [UfxDeviceCreate](#).

The client driver indicates completion of this event by calling the [UfxDeviceProprietaryChargerDetectComplete](#) method.

The client driver sends a request to the lower filter driver to determine if a proprietary charger is present. In response, the filter driver provides a GUID for each charger type it supports, and a list of that charger's properties. If a specific charger is configurable, the filter driver also provides a list of supported PropertyIDs and their possible values to configure the charger.

## Requirements

<b>Target Platform</b>	Windows
<b>Minimum KMDF version</b>	1.0
<b>Minimum UMDF version</b>	2.0
<b>Header</b>	ufxclient.h
<b>IRQL</b>	PASSIVE_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceProprietaryChargerDetectComplete](#)

# EVT\_UFX\_DEVICE\_PROPRIETARY\_CHARGER\_RESET callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to resets proprietary charger.

## Syntax

C++

```
EVT_UFX_DEVICE_PROPRIETARY_CHARGER_RESET  
EvtUfxDeviceProprietaryChargerReset;  
  
void EvtUfxDeviceProprietaryChargerReset(  
    [in] UFXDEVICE unnamedParam1  
)  
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

*EVT\_UFX\_DEVICE\_PROPRIETARY\_CHARGER\_RESET* is an optional event callback.

The USB function class extension (UFX) invokes this event callback to indicate that the USB device has been detached from the charger. The client driver initiates a request to the lower filter driver to reset the proprietary charger its initial state.

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceProprietaryChargerDetectComplete](#)

# EVT\_UFX\_DEVICE\_PROPRIETARY\_CHARGER\_SET\_PROPERTY callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to set charger information that it uses to enable charging over USB.

## Syntax

C++

```
EVT_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY
EvtUfxDeviceProprietaryChargerSetProperty;

void EvtUfxDeviceProprietaryChargerSetProperty(
    [in] UFXDEVICE unnamedParam1,
    [in] WDFREQUEST unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

The handle framework request object that for an [IOCTL\\_INTERNAL\\_CONFIGURE\\_CHARGER\\_PROPERTY](#) request from the battery minidriver.

## Return value

None

## Remarks

`EVT_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY` is an optional event callback.

*WdfRequest* is contains a request for [IOCTL\\_INTERNAL\\_CONFIGURE\\_CHARGER\\_PROPERTY](#), which specifies a charger ID that is known by the client driver and battery miniclass driver, and a voltage value in millivolts. The client driver can use this information to enable charging over the USB port at an appropriate current/voltage level.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceProprietaryChargerDetectComplete](#)

# EVT\_UFX\_DEVICE\_REMOTE\_WAKEUP\_SIG NAL callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to initiate remote wake-up on the function controller.

## Syntax

C++

```
EVT_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL EvtUfxDeviceRemoteWakeupsignal;

void EvtUfxDeviceRemoteWakeupsignal(
    [in] UFXDEVICE unnamedParam1
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#) method.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_REMOTE\_WAKEUP\_SIGNAL* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

C++

```
EVT_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL UfxDevice_EvtDeviceRemoteWakeupsignal;
```

```
VOID  
UfxDevice_EvtDeviceRemoteWakeupsignal (  
    _In_ UFXDEVICE UfxDevice  
)  
/*++  
Routine Description:
```

Signals Remote Wakeup to the Host by issuing a link state change command. It acquires and releases the power reference to ensure a valid power state before accessing the device.

Arguments:

UfxDevice - UFXDEVICE object representing the device.

```
--*/  
{  
NTSTATUS Status;  
PUFXDEVICE_CONTEXT DeviceContext;  
  
PAGED_CODE();  
  
TraceEntry();  
  
DeviceContext = UfxDeviceGetContext(UfxDevice);  
  
//  
// Stop Idle to ensure the device is in working state  
//  
  
Status = WdfDeviceStopIdle(DeviceContext->FdoWdfDevice, TRUE);  
if (!NT_SUCCESS(Status)) {  
    TraceError("Failed to stop idle %!STATUS!", Status);  
    goto End;  
}  
  
//  
// Issue a Link State Change Request.  
//  
  
//  
// ##### TODO: Insert code to issue a link state change on the controller  
#####  
//  
  
WdfDeviceResumeIdle(DeviceContext->FdoWdfDevice);  
  
End:  
    UfxDeviceEventComplete(UfxDevice, Status);
```

```
    TraceExit();  
}
```

# Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

- [UfxDeviceCreate](#)
- [UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_SUPER\_SPEED\_POWER\_FEATURE callback function (ufxclient.h)

Article 02/22/2024

The client driver's implementation to set or clear the specified power feature on the function controller.

## Syntax

C++

```
EVT_UFX_DEVICE_SUPER_SPEED_POWER_FEATURE EvtUfxDeviceSuperSpeedPowerFeature;

void EvtUfxDeviceSuperSpeedPowerFeature(
    [in] UFXDEVICE unnamedParam1,
    [in] USHORT unnamedParam2,
    [in] BOOLEAN unnamedParam3
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

Feature selector for the power feature that is set or cleared, as defined in section 9.4 of the USB 3.0 Specification. The feature selector can be one of these values:

- U1\_ENABLE
- U2\_ENABLE

[in] unnamedParam3

If TRUE, set the feature.

If FALSE, clear the feature.

# Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_SUPER\_SPEED\_POWER\_FEATURE* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

This event callback is only required for controllers that support SuperSpeed operation.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_SUPER_SPEED_POWER_FEATURE
UfxDevice_EvtDeviceSuperSpeedPowerFeature;

VOID
UfxDevice_EvtDeviceSuperSpeedPowerFeature (
    _In_ UFXDEVICE Device,
    _In_ USHORT Feature,
    _In_ BOOLEAN Set
)
/*++

Routine Description:

    EvtDeviceSuperSpeedPowerFeature handler for the UFXDEVICE object.

    Handles a set or clear U1/U2 request from the host.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    Feature - Indicates the feature being set or cleared. Either U1 or U2
enable.

    Set - Indicates if the feature should be set or cleared

--*/
{
    TraceEntry();
}
```

```

if (Feature == USB FEATURE_U1_ENABLE) {
    if (Set == TRUE) {
        //
        // ##### TODO: Insert code to initiate U1 #####
        //
    } else {
        //
        // ##### TODO: Insert code to exit U1 #####
        //
    }
} else if (Feature == USB FEATURE_U2_ENABLE) {
    if (Set == TRUE) {
        //
        // ##### TODO: Insert code to initiate U2 #####
        //
    } else {
        //
        // ##### TODO: Insert code to exit U2 #####
        //
    }
} else {
    NT_ASSERT(FALSE);
}

UfxDeviceEventComplete(Device, STATUS_SUCCESS);
TraceExit();
}

```

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	<=DISPATCH_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_TEST\_MODE\_SET callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to set the test mode of the function controller.

## Syntax

C++

```
EVT_UFX_DEVICE_TEST_MODE_SET EvtUfxDeviceTestModeSet;

void EvtUfxDeviceTestModeSet(
    [in] UFXDEVICE unnamedParam1,
    [in] ULONG unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

Test mode selector value as defined by the USB 2.0 Specification. These values are defined in usbfnbase.h

- USB\_TEST\_MODE\_TEST\_J 0x01
- USB\_TEST\_MODE\_TEST\_K 0x02
- USB\_TEST\_MODE\_TEST\_SE0\_NAK 0x03
- USB\_TEST\_MODE\_TEST\_PACKET 0x04
- USB\_TEST\_MODE\_TEST\_FORCE\_ENABLE 0x05

## Return value

None

# Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_TEST\_MODE\_SET* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_TEST_MODE_SET UfxDevice_EvtDeviceTestModeSet;

VOID
UfxDevice_EvtDeviceTestModeSet (
    _In_ UFXDEVICE UfxDevice,
    _In_ ULONG TestMode
)
/*++

Routine Description:

    EvtDeviceTestModeSet handler for the UFXDEVICE object.

    Handles a set test mode request from the host.  Places the controller
into
    the specified test mode.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    TestMode - Test mode value.  See Section 7.1.20 of the USB 2.0
specification for definitions of
                each test mode.

--*/
{
    NTSTATUS Status;

    UNREFERENCED_PARAMETER(TestMode);

    TraceEntry();

    //
    // ##### TODO: Insert code to put the controller into the specified test
mode #####
    //
}
```

```
    Status = STATUS_SUCCESS;

    UfxDeviceEventComplete(UfxDevice, Status);
    TraceExit();
}
```

## Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	<=DISPATCH_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# EVT\_UFX\_DEVICE\_TESTHOOK callback function (ufxclient.h)

Article02/22/2024

This IOCTL code is not supported.

## Syntax

C++

```
EVT_UFX_DEVICE_TESTHOOK EvtUfxDeviceTesthook;

NTSTATUS EvtUfxDeviceTesthook(
    [in]    UFXDEVICE unnamedParam1,
    [in]    PVOID unnamedParam2,
    [in]    size_t unnamedParam3,
    [out]   PVOID unnamedParam4,
    [in]    size_t unnamedParam5
)
{...}
```

## Parameters

[in] unnamedParam1

[in] unnamedParam2

[in] unnamedParam3

[out] unnamedParam4

[in] unnamedParam5

## Requirements

  Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0

Requirement	Value
Minimum UMDF version	2.0
Header	ufxclient.h (include Ufxclient.h)
IRQL	<=DISPATCH_LEVEL

# EVT\_UFX\_DEVICE\_USB\_STATE\_CHANGE callback function (ufxclient.h)

Article02/22/2024

The client driver's implementation to update the state of the USB device.

## Syntax

C++

```
EVT_UFX_DEVICE_USB_STATE_CHANGE EvtUfxDeviceUsbStateChange;

void EvtUfxDeviceUsbStateChange(
    [in] UFXDEVICE unnamedParam1,
    [in] USBFN_DEVICE_STATE unnamedParam2
)
{...}
```

## Parameters

[in] unnamedParam1

The handle to a USB device object that the client driver received in a previous call to the [UfxDeviceCreate](#).

[in] unnamedParam2

A USBFN\_DEVICE\_STATE-typed flag that indicates state of the USB device.

## Return value

None

## Remarks

The client driver for the function host controller registers its *EVT\_UFX\_DEVICE\_USB\_STATE\_CHANGE* implementation with the USB function class extension (UFX) by calling the [UfxDeviceCreate](#) method.

UFX invokes this event callback to inform the client driver about the new state of the device.

The client driver indicates completion of this event by calling the [UfxDeviceEventComplete](#) method.

## Examples

```
EVT_UFX_DEVICE_USB_STATE_CHANGE UfxDevice_EvtDeviceUsbStateChange;

VOID
UfxDevice_EvtDeviceUsbStateChange (
    _In_ UFXDEVICE UfxDevice,
    _In_ USBFN_DEVICE_STATE NewState
)
/*++

Routine Description:

    EvtDeviceUsbStateChange handler for the UFXDEVICE object.

Arguments:

    UfxDevice - UFXDEVICE object representing the device.

    NewState - The new device state.

--*/
{
    NTSTATUS Status;
    PUFXDEVICE_CONTEXT Context;
    PCONTROLLER_CONTEXT ControllerContext;
    ULONG EpIndex;
    USBFN_DEVICE_STATE OldState;

    PAGED_CODE();

    TraceEntry();

    Context = UfxDeviceGetContext(UfxDevice);
    ControllerContext = DeviceGetControllerContext(Context->FdowdfDevice);
    OldState = Context->UsbState;

    TraceInformation("New STATE: %d", NewState);

    Status = UfxDeviceStopOrResumeIdle(UfxDevice, NewState, Context-
>UsbPort);
    LOG_NT_MSG(Status, "Failed to stop or resume idle");
```

```

        WdfWaitLockAcquire(ControllerContext->InitializeDefaultEndpointLock,
NULL);
        if (ControllerContext->InitializeDefaultEndpoint == TRUE) {
            //
            // Reset endpoint 0. This is the last part of soft reset, which was
postponed
            //
            // until now, since we need to make sure EP0 is created by UFX.
            //
            DeviceInitializeDefaultEndpoint(Context->FdoWdfDevice);
            ControllerContext->InitializeDefaultEndpoint = FALSE;
        }
        WdfWaitLockRelease(ControllerContext->InitializeDefaultEndpointLock);

        if (NewState == UsbfnDeviceStateConfigured && OldState != UsbfnDeviceStateSuspended) {

            for (EpIndex = 1; EpIndex < WdfCollectionGetCount(Context-
>Endpoints); EpIndex++) {
                UfxEndpointConfigure(WdfCollectionGetItem(Context->Endpoints,
EpIndex));
            }

            //
            // ##### TODO: Insert code to allow the controller to accept U1/U2,
if supported #####
            //
        }

        if (NewState == UsbfnDeviceStateDetached) {
            KeSetEvent(&ControllerContext->DetachEvent,
IO_NO_INCREMENT,
FALSE);
        }

        UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);
        TraceExit();
    }
}

```

## Requirements

[\[\] Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0

Requirement	Value
Minimum UMDF version	2.0
Header	ufxclient.h
IRQL	PASSIVE_LEVEL

## See also

[UfxDeviceCreate](#)

[UfxDeviceEventComplete](#)

# UFX\_DEVICE\_CALLBACKS structure (ufxclient.h)

Article 02/22/2024

The **UFX\_DEVICE\_CALLBACKS** structure is used to define the event callback functions supported by the client driver.

## Syntax

C++

```
typedef struct _UFX_DEVICE_CALLBACKS {
    ULONG
    PFN_UFX_DEVICE_HOST_CONNECT
    PFN_UFX_DEVICE_HOST_DISCONNECT
    PFN_UFX_DEVICE_ADDRESSED
    PFN_UFX_DEVICE_ENDPOINT_ADD
    PFN_UFX_DEVICE_DEFAULT_ENDPOINT_ADD
    EvtDeviceDefaultEndpointAdd;
    PFN_UFX_DEVICE_USB_STATE_CHANGE
    PFN_UFX_DEVICE_PORT_CHANGE
    PFN_UFX_DEVICE_PORT_DETECT
    PFN_UFX_DEVICE_REMOTE_WAKEUP_SIGNAL
    EvtDeviceRemoteWakeupSignal;
    PFN_UFX_DEVICE_CONTROLLER_RESET
    PFN_UFX_DEVICE_TEST_MODE_SET
    PFN_UFX_DEVICE_TESTHOOK
    PFN_UFX_DEVICE_SUPER_SPEED_POWER_FEATURE
    EvtDeviceSuperSpeedPowerFeature;
    PFN_UFX_DEVICE_PROPRIETARY_CHARGER_DETECT
    EvtDeviceProprietaryChargerDetect;
    PFN_UFX_DEVICE_PROPRIETARY_CHARGER_SET_PROPERTY
    EvtDeviceProprietaryChargerSetProperty;
    PFN_UFX_DEVICE_PROPRIETARY_CHARGER_RESET
    EvtDeviceProprietaryChargerReset;
} UFX_DEVICE_CALLBACKS, *PUFX_DEVICE_CALLBACKS;
```

## Members

Size

The size of the **UFX\_DEVICE\_CALLBACKS** structure.

EvtDeviceHostConnect

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_HOST\\_CONNECT](#) callback routine.

`EvtDeviceHostDisconnect`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_HOST\\_DISCONNECT](#) callback routine.

`EvtDeviceAddressed`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_ADDRESSED](#) callback routine.

`EvtDeviceEndpointAdd`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_ENDPOINT\\_ADD](#) callback routine.

`EvtDeviceDefaultEndpointAdd`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) callback routine.

`EvtDeviceUsbStateChange`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_USB\\_STATE\\_CHANGE](#) callback routine.

`EvtDevicePortChange`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_PORT\\_CHANGE](#) callback routine.

`EvtDevicePortDetect`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_PORT\\_DETECT](#) callback routine.

`EvtDeviceRemoteWakeupSignal`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_REMOTE\\_WAKEUP\\_SIGNAL](#) callback routine.

`EvtDeviceControllerReset`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_CONTROLLER\\_RESET](#) callback routine.

`EvtDeviceTestModeSet`

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_TEST\\_MODE\\_SET](#) callback routine.

`EvtDeviceTestHook`

Reserved. Should be set to NULL.

#### **EvtDeviceSuperSpeedPowerFeature**

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_SUPER\\_SPEED\\_POWER\\_FEATURE](#) callback routine.

#### **EvtDeviceProprietaryChargerDetect**

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_DETECT\\_PROPRIETARY\\_CHARGER](#) callback routine.

#### **EvtDeviceProprietaryChargerSetProperty**

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_SET\\_PROPERTY](#) callback routine.

#### **EvtDeviceProprietaryChargerReset**

A pointer to the client driver's [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_RESET](#) callback routine.

## Remarks

The client driver shall use the [UFX\\_DEVICE\\_CALLBACKS\\_INIT](#) macro to initialize the [UFX\\_DEVICE\\_CALLBACKS](#) structure, and then shall set fields of structure to the appropriate event callback routines prior to calling the [UfxDeviceCreate](#) export function.

## Requirements

 Expand table

Requirement	Value
Header	ufxclient.h

# UFX\_DEVICE\_CALLBACKS\_INIT function (ufxclient.h)

Article02/22/2024

The **UFX\_DEVICE\_CALLBACKS\_INIT** macro initializes the [UFX\\_DEVICE\\_CALLBACKS](#) structure.

## Syntax

C++

```
void UFX_DEVICE_CALLBACKS_INIT(
    [out] PUFX_DEVICE_CALLBACKS Callbacks
);
```

## Parameters

[out] **Callbacks**

A pointer to the [UFX\\_DEVICE\\_CALLBACKS](#) structure.

## Return value

None

## Remarks

The **UFX\_DEVICE\_CALLBACKS\_INIT** macro will set all fields of the [UFX\\_DEVICE\\_CALLBACKS](#) structure to zero and set the **Size** field appropriately.

The client driver uses the **UFX\_DEVICE\_CALLBACKS\_INIT** macro to initialize the [UFX\\_DEVICE\\_CALLBACKS](#) structure prior to calling [UfxDeviceCreate](#).

## Requirements

[] Expand table

Requirement	Value
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib

# UFX\_DEVICE\_CAPABILITIES\_INIT function (ufxclient.h)

Article02/22/2024

The **UFX\_DEVICE\_CAPABILITIES\_INIT** macro initializes the [UFX\\_DEVICE\\_CAPABILITIES](#) structure.

## Syntax

C++

```
void UFX_DEVICE_CAPABILITIES_INIT(
    [out] PUFX_DEVICE_CAPABILITIES Capabilities
);
```

## Parameters

[out] Capabilities

Pointer to the [UFX\\_DEVICE\\_CAPABILITIES](#) structure.

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib

# UFX\_ENDPOINT\_CALLBACKS structure (ufxclient.h)

Article 02/22/2024

The **UFX\_ENDPOINT\_CALLBACKS** structure is used to define the event callback functions supported by the client driver.

## Syntax

C++

```
typedef struct _UFX_ENDPOINT_CALLBACKS {
    ULONG Size;
} UFX_ENDPOINT_CALLBACKS, *PUFX_ENDPOINT_CALLBACKS;
```

## Members

Size

The size of the **UFX\_ENDPOINT\_CALLBACKS** structure.

## Requirements

  Expand table

Requirement	Value
Header	ufxclient.h

# UFX\_ENDPOINT\_CALLBACKS\_INIT function (ufxclient.h)

Article02/22/2024

The **UFX\_ENDPOINT\_CALLBACKS\_INIT** macro initializes the [UFX\\_ENDPOINT\\_CALLBACKS](#) structure.

## Note

Note that there are currently no endpoint callback functions defined in the **UFX\_ENDPOINT\_CALLBACKS** structure.

## Syntax

C++

```
void UFX_ENDPOINT_CALLBACKS_INIT(
    [out] PUFX_ENDPOINT_CALLBACKS Callbacks
);
```

## Parameters

[out] Callbacks

A pointer to the [UFX\\_ENDPOINT\\_CALLBACKS](#) structure.

## Return value

None

## Remarks

The **UFX\_ENDPOINT\_CALLBACKS\_INIT** macro will set all fields of the [UFX\\_ENDPOINT\\_CALLBACKS](#) structure to zero and set the size field appropriately.

**Note** Note that there are currently no endpoint callback functions defined in the **UFX\_ENDPOINT\_CALLBACKS** structure.

# Requirements

[Expand table](#)

Requirement	Value
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib

# UfxDeviceCreate function (ufxclient.h)

Article10/21/2021

Creates a UFX device object, registers event callback routines, and specifies capabilities specific to the controller.

## Syntax

C++

```
NTSTATUS UfxDeviceCreate(
    WDFDEVICE                WdfDevice,
    [in]          PUFX_DEVICE_CALLBACKS   Callbacks,
    [in]          PUFX_DEVICE_CAPABILITIES Capabilities,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [out]           UFXDEVICE            *UfxDevice
);
```

## Parameters

`WdfDevice`

A handle to a WDF device object.

`[in] Callbacks`

A structure of type [UFX\\_DEVICE\\_CALLBACKS](#) that contains pointers to driver-supplied callback routines to be associated with the UFX device object.

`[in] Capabilities`

A pointer to a [UFX\\_DEVICE\\_CAPABILITIES](#) structure.

`[in, optional] Attributes`

A pointer to a [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that contains driver-supplied attributes for the new object. This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

`[out] UfxDevice`

A pointer to a location that receives a handle to the new UFX device object.

# Return value

If the operation is successful, the method returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The client driver must call [WdfDeviceCreate](#) before it calls [UfxDeviceCreate](#). Typically, the client driver calls [UfxDeviceCreate](#) from its [EvtDriverDeviceAdd](#) callback routine.

The following code snippet shows how to call [UfxDeviceCreate](#).

```
NTSTATUS
UfxDevice_DeviceCreate (
    _In_ WDFDEVICE WdfDevice
)
/*++

Routine Description:

    Routine that registers with UFX and creates the UFX device object.

Arguments:

    WdfDevice - Wdf object representing the device.

Return Value:

    NTSTATUS.

--*/
{
    NTSTATUS Status;
    PCONTROLLER_CONTEXT ControllerContext;
    UFX_DEVICE_CALLBACKS UfxDeviceCallbacks;
    UFX_DEVICE_CAPABILITIES UfxDeviceCapabilities;
    PUFXDEVICE_CONTEXT UfxDeviceContext;
    UFXDEVICE UfxDevice;
    WDF_OBJECT_ATTRIBUTES Attributes;

    PAGED_CODE();

    TraceEntry();

    ControllerContext = DeviceGetControllerContext(WdfDevice);

    UFX_DEVICE_CAPABILITIES_INIT(&UfxDeviceCapabilities);
```

```

UfxDeviceCapabilities.MaxSpeed = UsbSuperSpeed;
UfxDeviceCapabilities.RemoteWakeSignalDelay =
REMOTE_WAKEUP_TIMEOUT_INTERVAL_MS;

//
// Set bitmasks that define the IN and OUT endpoint numbers that are
available on the controller
//

//
// ##### TODO: Set the IN endpoint mask here if not all endpoint
addresses are supported #####
//

// For illustration purposes sample will set default control endpoint 0,
1-4, 8
UfxDeviceCapabilities.InEndpointBitmap = 0x011F;

//
// ##### TODO: Set the OUT endpoint mask here if not all endpoint
addresses are supported #####
//

// For illustration purposes sample will set default control endpoint 0,
2-7
//

UfxDeviceCapabilities.OutEndpointBitmap = 0x00FD;

//
// Set the event callbacks for the ufxdevice
//
UFX_DEVICE_CALLBACKS_INIT(&UfxDeviceCallbacks);
UfxDeviceCallbacks.EvtDeviceHostConnect =
UfxDevice_EvtDeviceHostConnect;
UfxDeviceCallbacks.EvtDeviceHostDisconnect =
UfxDevice_EvtDeviceHostDisconnect;
UfxDeviceCallbacks.EvtDeviceAddressed = UfxDevice_EvtDeviceAddressed;
UfxDeviceCallbacks.EvtDeviceEndpointAdd =
UfxDevice_EvtDeviceEndpointAdd;
UfxDeviceCallbacks.EvtDeviceDefaultEndpointAdd =
UfxDevice_EvtDeviceDefaultEndpointAdd;
UfxDeviceCallbacks.EvtDeviceUsbStateChange =
UfxDevice_EvtDeviceUsbStateChange;
UfxDeviceCallbacks.EvtDevicePortChange = UfxDevice_EvtDevicePortChange;
UfxDeviceCallbacks.EvtDevicePortDetect = UfxDevice_EvtDevicePortDetect;
UfxDeviceCallbacks.EvtDeviceRemoteWakeupsignal =
UfxDevice_EvtDeviceRemoteWakeupsignal;
UfxDeviceCallbacks.EvtDeviceTestModeSet =
UfxDevice_EvtDeviceTestModeSet;
UfxDeviceCallbacks.EvtDeviceSuperSpeedPowerFeature =
UfxDevice_EvtDeviceSuperSpeedPowerFeature;

// Context associated with UFXDEVICE object
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&Attributes, UFXDEVICE_CONTEXT);
Attributes.EvtCleanupCallback = UfxDevice_EvtCleanupCallback;

// Create the UFXDEVICE object

```

```
    Status = UfxDeviceCreate(WdfDevice,
        &UfxDeviceCallbacks,
        &UfxDeviceCapabilities,
        &Attributes,
        &UfxDevice);

    ...
}
```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	PASSIVE_LEVEL

# UfxDeviceEventComplete function (ufxclient.h)

Article 10/21/2021

Informs UFX that the client driver has completed processing a UFX callback function.

## Syntax

C++

```
void UfxDeviceEventComplete(
    [in] UFXDEVICE UfxDevice,
    [in] NTSTATUS Status
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] Status

Status of the event being completed.

## Return value

None

## Remarks

The client driver calls **UfxDeviceEventComplete** to signal completion of the following callback functions:

- [EVT\\_UFX\\_DEVICE\\_HOST\\_CONNECT](#)
- [EVT\\_UFX\\_DEVICE\\_HOST\\_DISCONNECT](#)
- [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_SET\\_PROPERTY](#)
- [EVT\\_UFX\\_DEVICE\\_PROPRIETARY\\_CHARGER\\_RESET](#)
- [EVT\\_UFX\\_DEVICE\\_ADDRESSED](#)

- [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#)
- [EVT\\_UFX\\_DEVICE\\_USB\\_STATE\\_CHANGE](#)
- [EVT\\_UFX\\_DEVICE\\_PORT\\_CHANGE](#)
- [EVT\\_UFX\\_DEVICE\\_REMOTE\\_WAKEUP\\_SIGNAL](#)
- [EVT\\_UFX\\_DEVICE\\_TEST\\_MODE\\_SET](#)
- [EVT\\_UFX\\_DEVICE\\_SUPER\\_SPEED\\_POWER\\_FEATURE](#)
- [EVT\\_UFX\\_DEVICE\\_CONTROLLER\\_RESET](#)

For example, your callback function could use the following code:

```
EventComplete = TRUE;  
  
...  
  
if (EventComplete) {  
    UfxDeviceEventComplete(UfxDevice, STATUS_SUCCESS);  
}
```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceIoControl function (ufxclient.h)

Article 02/22/2024

Passes non-internal IOCTLs from user-mode to UFX.

## Syntax

C++

```
BOOLEAN UfxDeviceIoControl(
    [in] UFXDEVICE UfxDevice,
    [in] WDFREQUEST Request,
    [in] size_t     OutputBufferLength,
    [in] size_t     InputBufferLength,
    [in] ULONG      IoControlCode
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] Request

A handle to a framework request object.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available.

[in] IoControlCode

The driver-defined or system-defined IOCTL that is associated with the request.

## Return value

A Boolean value indicating if the call was successful.

## Remarks

The client driver calls **UfxDeviceIoControl** to forward non-internal IOCTLs that it receives in its **EvtIoDeviceControl** callback function to UFX. The following example shows how:

C++

```
VOID
DefaultQueue_EvtIoDeviceControl(
    _In_ WDFQUEUE Queue,
    _In_ WDFREQUEST Request,
    _In_ size_t OutputBufferLength,
    _In_ size_t InputBufferLength,
    _In_ ULONG IoControlCode
)
/*++
```

**Routine Description:**

EvtIoDeviceControl handler for the default Queue

**Arguments:**

Queue - Handle to the framework queue object that is associated with the I/O request.

Request - Handle to a framework request object.

OutputBufferLength - Size of the output buffer in bytes

InputBufferLength - Size of the input buffer in bytes

IoControlCode - I/O control code.

--\*/

{

```
WDFDEVICE WdfDevice;
PCONTROLLER_CONTEXT ControllerContext;
BOOLEAN HandledbyUfx;
```

```
TraceEntry();
```

```
TraceVerbose("Queue 0x%p, Request 0x%p, OutputBufferLength %d, "
            "InputBufferLength %d, IoControlCode %d",
            Queue, Request, (int) OutputBufferLength,
            (int) InputBufferLength, IoControlCode);
```

```
WdfDevice = WdfIoQueueGetDevice(Queue);
ControllerContext = DeviceGetControllerContext(WdfDevice);
```

```

HandledbyUfx = UfxDeviceIoControl(
    ControllerContext->UfxDevice,
    Request,
    OutputBufferLength,
    InputBufferLength,
    IoControlCode);

if (!HandledbyUfx) {
    TraceError("Recieved an unsupported IOCTL");
    WdfRequestComplete(Request, STATUS_INVALID_DEVICE_REQUEST);
}

TraceExit();
}

```

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceIoInternalControl function (ufxclient.h)

Article 02/22/2024

Passes kernel mode IOCTLs to UFX.

## Syntax

C++

```
BOOLEAN UfxDeviceIoInternalControl(
    [in] UFXDEVICE UfxDevice,
    [in] WDFREQUEST Request,
    [in] size_t     OutputBufferLength,
    [in] size_t     InputBufferLength,
    [in] ULONG      IoControlCode
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] Request

A handle to a framework request object.

[in] OutputBufferLength

The length, in bytes, of the request's output buffer, if an output buffer is available.

[in] InputBufferLength

The length, in bytes, of the request's input buffer, if an input buffer is available.

[in] IoControlCode

The driver-defined or system-defined IOCTL that is associated with the request.

## Return value

A Boolean value indicating if the call was successful.

## Remarks

The client driver calls [UfxDeviceIoInternalControl](#) to forward internal IOCTLs that it receives in its [EvtIoInternalDeviceControl](#) callback function to UFX. The following example shows how:

C++

```
VOID
DefaultQueue_EvtIoInternalDeviceControl(
    _In_ WDFQUEUE Queue,
    _In_ WDFREQUEST Request,
    _In_ size_t OutputBufferLength,
    _In_ size_t InputBufferLength,
    _In_ ULONG IoControlCode
)
/*++

Routine Description:

    EvtIoInternalDeviceControl handler for the default Queue

Arguments:

    Queue - Handle to the framework queue object that is associated with
            the
            I/O request.

    Request - Handle to a framework request object.

    OutputBufferLength - Size of the output buffer in bytes

    InputBufferLength - Size of the input buffer in bytes

    IoControlCode - I/O control code.

--*/
{
    WDFDEVICE WdfDevice;
    PCONTROLLER_CONTEXT ControllerContext;
    BOOLEAN HandledbyUfx;

    TraceEntry();

    TraceVerbose("InternalQueue 0x%p, Request 0x%p, OutputBufferLength %d, "
                "InputBufferLength %d, IoControlCode %d",
                Queue, Request, (int)OutputBufferLength,
                (int)InputBufferLength, IoControlCode);
```

```

WdfDevice = WdfIoQueueGetDevice(Queue);
ControllerContext = DeviceGetControllerContext(WdfDevice);

HandledbyUfx = UfxDeviceIoInternalControl(
    ControllerContext->UfxDevice,
    Request,
    OutputBufferLength,
    InputBufferLength,
    IoControlCode);

if (!HandledbyUfx) {
    TraceError("Received an un supported IOCTL");
    WdfRequestComplete(Request, STATUS_INVALID_DEVICE_REQUEST);
}

TraceExit();
}

```

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyAttach function (ufxclient.h)

Article 09/19/2022

Notifies UFX that the device's USB cable has been attached.

## Syntax

C++

```
void UfxDeviceNotifyAttach(
    [in] UFXDEVICE UfxDevice
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Remarks

When the client driver calls [UfxDeviceNotifyAttach](#), the USB function class extension (UFX) does the following:

- Moves the device to the powered state, as defined in the USB specification.
- Allows device enumeration to occur.

The client driver typically calls [UfxDeviceNotifyAttach](#) from its [EVT\\_WDF\\_INTERRUPT\\_DPC](#) callback function, as shown in the following example.

C++

```
VOID
DeviceInterrupt_EvtInterruptDpc (
    _In_ WDF_INTERRUPT Interrupt,
```

```

    _In_ WDFOBJECT AssociatedObject
)
/*++

Routine Description:

    'EVT_WDF_INTERRUPT_DPC' handler for the device interrupt object.

Arguments:

    Interrupt - Associated interrupt object.

    AssociatedObject - FDO Object

--*/
{
    WDFDEVICE WdfDevice;
    PDEVICE_INTERRUPT_CONTEXT InterruptContext;
    PCONTROLLER_CONTEXT ControllerContext;
    BOOLEAN Attached;
    BOOLEAN GotAttachOrDetach;
    CONTROLLER_EVENT ControllerEvent;

    UNREFERENCED_PARAMETER(Interrupt);

    TraceEntry();

    WdfDevice = (WDFDEVICE) AssociatedObject;
    ControllerContext = DeviceGetControllerContext(WdfDevice);

    WdfSpinLockAcquire(ControllerContext->DpcLock);

    WdfInterruptAcquireLock(ControllerContext->DeviceInterrupt);
    Attached = ControllerContext->Attached;
    GotAttachOrDetach = ControllerContext->GotAttachOrDetach;
    ControllerContext->GotAttachOrDetach = FALSE;
    WdfInterruptReleaseLock(ControllerContext->DeviceInterrupt);

    //
    // Handle attach/detach events
    //
    if (GotAttachOrDetach) {
        if (Attached && ControllerContext->WasAttached) {
            //
            // We must have gotten at least one detach. Need to reset the
            state.
            //
            ControllerContext->RemoteWakeUpRequested = FALSE;
            ControllerContext->Suspended = FALSE;
            UfxDeviceNotifyDetach(ControllerContext->UfxDevice);
        }

        if (Attached) {
            ControllerContext->RemoteWakeUpRequested = FALSE;
            ControllerContext->Suspended = FALSE;
        }
    }
}

```

```

        UfxDeviceNotifyAttach(ControllerContext->UfxDevice);
    }

}

ControllerContext->WasAttached = Attached;

InterruptContext = DeviceInterruptGetContext(ControllerContext-
>DeviceInterrupt);

//  

// ##### TODO: Insert code to read and dispatch events from the  

controller #####  

//  

// The sample will assume an endpoint event of  

EndpointEventTransferComplete  

ControllerEvent.Type = EventTypeEndpoint;  

ControllerEvent.u.EndpointEvent = EndpointEventTransferComplete;  

//  

// Handle events from the controller  

//  

switch (ControllerEvent.Type) {  

case EventTypeDevice:  

    HandleDeviceEvent(WdfDevice, ControllerEvent.u.DeviceEvent);  

    break;  

  

case EventTypeEndpoint:  

    HandleEndpointEvent(WdfDevice, ControllerEvent.u.EndpointEvent);  

    break;  

}  

  

WdfSpinLockRelease(ControllerContext->DpcLock);  

  

TraceExit();
}

```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyDetach function (ufxclient.h)

Article 02/22/2024

Notifies UFX that the device's USB cable has been detached.

## Syntax

C++

```
void UfxDeviceNotifyDetach(
    [in] UFXDEVICE UfxDevice
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Remarks

This method is called by the client driver when it receives a USB cable detach event. Once the detach event is processed, all endpoints should be disabled and device should move to a low power mode.

The client driver typically calls [UfxDeviceNotifyDetach](#) from its [EVT\\_WDF\\_INTERRUPT\\_DPC](#) callback function, as shown in the following example.

C++

```
VOID
DeviceInterrupt_EvtInterruptDpc (
    _In_ WDFINTERRUPT Interrupt,
    _In_ WDOBJECT AssociatedObject
)
```

```

/*++

Routine Description:

    'EVT_WDF_INTERRUPT_DPC' handler for the device interrupt object.

Arguments:

    Interrupt - Associated interrupt object.

    AssociatedObject - FDO Object

--*/
{

    WDFDEVICE WdfDevice;
    PDEVICE_INTERRUPT_CONTEXT InterruptContext;
    PCONTROLLER_CONTEXT ControllerContext;
    BOOLEAN Attached;
    BOOLEAN GotAttachOrDetach;
    CONTROLLER_EVENT ControllerEvent;

    UNREFERENCED_PARAMETER(Interrupt);

    TraceEntry();

    WdfDevice = (WDFDEVICE) AssociatedObject;
    ControllerContext = DeviceGetControllerContext(WdfDevice);

    WdfSpinLockAcquire(ControllerContext->DpcLock);

    WdfInterruptAcquireLock(ControllerContext->DeviceInterrupt);
    Attached = ControllerContext->Attached;
    GotAttachOrDetach = ControllerContext->GotAttachOrDetach;
    ControllerContext->GotAttachOrDetach = FALSE;
    WdfInterruptReleaseLock(ControllerContext->DeviceInterrupt);

    //

    // Handle attach/detach events
    //
    if (GotAttachOrDetach) {
        if (Attached && ControllerContext->WasAttached) {
            //
            // We must have gotten at least one detach. Need to reset the
            state.
            //
            ControllerContext->RemoteWakeUpRequested = FALSE;
            ControllerContext->Suspended = FALSE;
            UfxDeviceNotifyDetach(ControllerContext->UfxDevice);
        }

        if (Attached) {
            ControllerContext->RemoteWakeUpRequested = FALSE;
            ControllerContext->Suspended = FALSE;
            UfxDeviceNotifyAttach(ControllerContext->UfxDevice);
        }
    }
}

```

```

}

ControllerContext->WasAttached = Attached;

InterruptContext = DeviceInterruptGetContext(ControllerContext-
>DeviceInterrupt);

// 
// ##### TODO: Insert code to read and dispatch events from the
controller #####
//

// The sample will assume an endpoint event of
EndpointEventTransferComplete
ControllerEvent.Type = EventTypeEndpoint;
ControllerEvent.u.EndpointEvent = EndpointEventTransferComplete;

//
// Handle events from the controller
//
switch (ControllerEvent.Type) {
case EventTypeDevice:
    HandleDeviceEvent(WdfDevice, ControllerEvent.u.DeviceEvent);
    break;

case EventTypeEndpoint:
    HandleEndpointEvent(WdfDevice, ControllerEvent.u.EndpointEvent);
    break;
}

WdfSpinLockRelease(ControllerContext->DpcLock);

TraceExit();
}

```

## Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyFinalExit function (ufxclient.h)

Article02/22/2024

Notifies UFX that the device is detached.

## Syntax

C++

```
void UfxDeviceNotifyFinalExit(
    UFXDEVICE UfxDevice
);
```

## Parameters

UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1803
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyHardwareFailure function (ufxclient.h)

Article02/22/2024

Notifies UFX about a non-recoverable hardware failure in the controller.

## Syntax

C++

```
void UfxDeviceNotifyHardwareFailure(
    [in]          UFXDEVICE                  UfxDevice,
    [in, optional] PUFX_HARDWARE_FAILURE_CONTEXT HardwareFailureContext
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in, optional] HardwareFailureContext

Optional pointer to a client driver-defined [UFX\\_HARDWARE\\_FAILURE\\_CONTEXT](#) structure containing controller-specific information about the hardware failure.

## Return value

None

## Remarks

The client driver calls [UfxDeviceNotifyHardwareFailure](#) when the controller has entered a non-recoverable hardware failure (such as PHY lockup). UFX can try resetting the controller to see if the controller can be recovered. The following example shows the syntax for the call:

```
UfxDeviceNotifyHardwareFailure(  
    ControllerContext->UfxDevice,  
    (PUFX_HARDWARE_FAILURE_CONTEXT) HardwareFailureContext);
```

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyHardwareReady function (ufxclient.h)

Article02/22/2024

Notifies UFX that the hardware is ready.

## Syntax

C++

```
void UfxDeviceNotifyHardwareReady(
    [in] UFXDEVICE UfxDevice
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Remarks

The client driver typically calls [UfxDeviceNotifyHardwareReady](#) from its [EvtDeviceD0Entry](#) callback function, as shown in the following example.

```
NTSTATUS
OnEvtDeviceD0Entry (
    _In_ WDFDEVICE Device,
    _In_ WDF_POWER_DEVICE_STATE PreviousState
)
/*++
```

Routine Description:

Called by the framework after entering D0 state.

Arguments:

Device - WDFDEVICE framework handle to the bus FDO.

PreviousState - The WDF\_POWER\_DEVICE\_STATE from which the stack is making this transition.

Return Value:

Returns STATUS\_SUCCESS or an appropriate NTSTATUS code otherwise.

```
--*/
{
    PCONTROLLER_CONTEXT ControllerContext;

    TraceEntry();

    ControllerContext = DeviceGetControllerContext(Device);

    if (PreviousState > WdfPowerDeviceD1) {
        DevicePerformSoftReset(Device);

        WdfWaitLockAcquire(ControllerContext->InitializeDefaultEndpointLock,
NULL);
        ControllerContext->InitializeDefaultEndpoint = TRUE;
        WdfWaitLockRelease(ControllerContext-
>InitializeDefaultEndpointLock);
    }

    if (PreviousState == WdfPowerDeviceD3Final) {
        //
        // Notify UFX that HW is now ready
        //
        UfxDeviceNotifyHardwareReady(ControllerContext->UfxDevice);
    }

    TraceExit();
    return STATUS_SUCCESS;
}
```

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows

Requirement	Value
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyReset function (ufxclient.h)

Article 02/22/2024

Notifies UFX about a USB bus reset event.

## Syntax

C++

```
void UfxDeviceNotifyReset(
    [in] UFXDEVICE          UfxDevice,
    [in] USB_DEVICE_SPEED  DeviceSpeed
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] DeviceSpeed

Contains a value of type [USB\\_DEVICE\\_SPEED](#) that indicates the speed of the device.

## Return value

None

## Remarks

The client driver calls [UfxDeviceNotifyReset](#) when it receives a bus reset event. All non-default endpoints should be disabled and the default endpoint should be reset. The device moves to the default state.

The client driver typically calls [UfxDeviceNotifyReset](#) from its [EvtInterruptDpc](#) callback function. The following example shows how to handle a reset event.

```

VOID
HandleUsbConnect (
    WDFDEVICE WdfDevice
)
/*++

Routine Description:

    Handles a connect event from the controller.

Arguments:

    WdfDevice - WDFDEVICE object representing the controller.

--*/
{
    PCONTROLLER_CONTEXT ControllerContext;
    USB_DEVICE_SPEED DeviceSpeed;

    TraceEntry();

    ControllerContext = DeviceGetControllerContext(WdfDevice);

    //
    // Read the device speed.
    //

    //
    // ##### TODO: Add code to read device speed from the controller #####
    //

    // Sample will assume SuperSpeed operation for illustration purposes
    DeviceSpeed = UsbSuperSpeed;

    //
    // ##### TODO: Add any code needed to configure the controller after
    connect has occurred #####
    //

    ControllerContext->Speed = DeviceSpeed;
    TraceInformation("Connected Speed is %d!", DeviceSpeed);

    //
    // Notify UFX about reset, which will take care of updating
    // Max Packet Size for EP0 by calling descriptor update.
    //
    UfxDeviceNotifyReset(ControllerContext->UfxDevice, DeviceSpeed);

    ControllerContext->Connect = TRUE;
}

```

```
    TraceExit();  
}
```

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifyResume function (ufxclient.h)

Article 02/22/2024

Notifies UFX about a USB bus resume event.

## Syntax

C++

```
void UfxDeviceNotifyResume(
    [in] UFXDEVICE UfxDevice
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Remarks

The client driver calls **UfxDeviceNotifyResume** when it receives a bus resume event. The controller should return to the same state it was in at the time of the bus resume event.

The client driver typically calls **UfxDeviceNotifyResume** from its [EvtInterruptDpc](#) callback function. The following example shows how to handle a resume event.

```
case DeviceEventWakeUp:
    if (ControllerContext->Suspended) {
        ControllerContext->Suspended = FALSE;
        UfxDeviceNotifyResume(ControllerContext->UfxDevice);
    }
```

```
break;
```

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceNotifySuspend function (ufxclient.h)

Article 02/22/2024

Notifies UFX about a USB bus suspend event.

## Syntax

C++

```
void UfxDeviceNotifySuspend(
    [in] UFXDEVICE UfxDevice
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

## Return value

None

## Remarks

The client driver calls **UfxDeviceNotifySuspend** when it receives a bus suspend event. The default endpoint should be reset on a bus suspend. The device should move to a low power mode.

The client driver typically calls **UfxDeviceNotifySuspend** from its [EvtInterruptDpc](#) callback function. The following example shows how to handle a suspend event.

```
case DeviceEventSuspend:
    if (!ControllerContext->Suspended) {
        ControllerContext->Suspended = TRUE;
        UfxDeviceNotifySuspend(ControllerContext->UfxDevice);
    }
```

```
break;
```

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDevicePortDetectComplete function (ufxclient.h)

Article 10/21/2021

Notifies UFX about the port type that was detected.

## Syntax

C++

```
void UfxDevicePortDetectComplete(
    [in] UFXDEVICE          UfxDevice,
    [in] USBFN_PORT_TYPE   PortType
);
```

## Parameters

[in] `UfxDevice`

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] `PortType`

Contains an enumeration value of type [USBFN\\_PORT\\_TYPE](#).

## Return value

None

## Remarks

The client driver calls `UfxDevicePortDetectComplete` when port detection is complete. On some platforms, UFX may use the reported port type to notify the battery manager of the maximum current it can draw from the USB port.

The client driver typically calls `UfxDevicePortDetectComplete` from its [EVT\\_UFX\\_DEVICE\\_PORT\\_DETECT](#) callback function, as shown in this example.

```
// In this example we will return an unknown port type. This will allow
UFX to connect to a host if
    // one is present. UFX will timeout after 5 seconds if no host is
    // present and transition to
        // an invalid charger type, which will allow the controller to exit D0.
        //
    UfxDevicePortDetectComplete(ControllerContext->UfxDevice,
UsbfnUnknownPort);
```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDevicePortDetectCompleteEx function (ufxclient.h)

Article 02/22/2024

Notifies UFX about the port type that was detected, and optionally requests an action.

## Syntax

C++

```
void UfxDevicePortDetectCompleteEx(
    [in] UFXDEVICE          UfxDevice,
    [in] USBFN_PORT_TYPE   PortType,
    [in] USBFN_ACTION      Action
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] PortType

Contains an enumeration value of type [USBFN\\_PORT\\_TYPE](#).

[in] Action

Contains an enumeration value of type [USBFN\\_ACTION](#).

## Return value

None

## Remarks

The client driver calls [UfxDevicePortDetectCompleteEx](#) after port detection is complete, typically from its [EVT\\_UFX\\_DEVICE\\_PORT\\_DETECT](#) callback function. On some platforms, UFX may use the reported port type to notify the battery manager of the maximum current it can draw from the USB port.

If the *Action* parameter is set to **UsbfnActionNoCad**, UFX does not notify the battery manager at all.

If the *Action* parameter is set to **UsbfnActionDetectProprietaryCharger**, UFX requests that the client driver initiate proprietary charger detection by calling the client driver's **EVT\_UFX\_DEVICE\_DETECT\_PROPRIETARY\_CHARGER** callback function.

The following snippet shows how a client driver calls **UfxDevicePortDetectCompleteEx**.

```
switch (OnAttach.AttachAction) {
    case UsbfnPortDetected:
        TraceInformation("Port Detected");
        UfxDevicePortDetectComplete(
            ControllerData->UfxDevice,
            OnAttach.PortType);
        break;

    case UsbfnPortDetectedNoCad:
        TraceInformation("Port Detected No CAD");
        UfxDevicePortDetectCompleteEx(
            ControllerData->UfxDevice,
            OnAttach.PortType,
            UsbfnActionNoCad);
        break;
}
```

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxDeviceProprietaryChargerDetectComplete function (ufxclient.h)

Article02/22/2024

Notifies UFX about a detected proprietary port/charger type.

## Syntax

C++

```
void UfxDeviceProprietaryChargerDetectComplete(
    [in] UFXDEVICE           UfxDevice,
    [in] PUFX_PROPRIETARY_CHARGER DetectedCharger
);
```

## Parameters

[in] UfxDevice

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] DetectedCharger

A pointer to a [UFX\\_PROPRIETARY\\_CHARGER](#) structure.

## Return value

None

## Remarks

The client driver calls [UfxDeviceProprietaryChargerDetectComplete](#) after attempting to detect a proprietary charger on the upstream port, typically from within its [EvtDriverDeviceAdd](#) callback function.

Do not call [UfxDeviceProprietaryChargerDetectComplete](#) before UFX calls the client driver's [EVT\\_UFX\\_DEVICE\\_DETECT\\_PROPRIETARY\\_CHARGER](#) callback function.

The following snippet shows how a client driver calls [UfxDeviceProprietaryChargerDetectComplete](#):

```
UfxDeviceProprietaryChargerDetectComplete(  
    ChargerContext->UfxDevice,  
    &pControllerData->DetectedCharger);
```

# Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	PASSIVE_LEVEL

# UfxEndpointCreate function (ufxclient.h)

Article10/21/2021

Creates an endpoint object.

## Syntax

C++

```
NTSTATUS UfxEndpointCreate(
    [in]          UFXDEVICE             UfxDevice,
    [in, out]      PUFXENDPOINT_INIT   EndpointInit,
    [in, optional] PWDF_OBJECT_ATTRIBUTES Attributes,
    [in]          PWDF_IO_QUEUE_CONFIG TransferQueueConfig,
    [in, optional] PWDF_OBJECT_ATTRIBUTES TransferQueueAttributes,
    [in]          PWDF_IO_QUEUE_CONFIG CommandQueueConfig,
    [in, optional] PWDF_OBJECT_ATTRIBUTES CommandQueueAttributes,
    [out]         UFXENDPOINT          *UfxEndpoint
);
```

## Parameters

[in] `UfxDevice`

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in, out] `EndpointInit`

Opaque structure passed by UFX in the call to [EVT\\_UFX\\_DEVICE\\_ENDPOINT\\_ADD](#) or [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#).

[in, optional] `Attributes`

A pointer to caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure. This structure must be initialized with [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT](#) or [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT\\_CONTEXT\\_TYPE](#). This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

[in] `TransferQueueConfig`

A pointer to a caller allocated [WDF\\_IO\\_QUEUE\\_CONFIG](#) structure. This structure must be initialized with [WDF\\_IO\\_QUEUE\\_CONFIG\\_INIT](#).

[in, optional] TransferQueueAttributes

A pointer to caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure. This structure must be initialized with [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT](#) or [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT\\_CONTEXT\\_TYPE](#). This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

[in] CommandQueueConfig

A pointer to a caller allocated [WDF\\_IO\\_QUEUE\\_CONFIG](#) structure. This structure must be initialized with [WDF\\_IO\\_QUEUE\\_CONFIG\\_INIT](#).

[in, optional] CommandQueueAttributes

A pointer to caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure. This structure must be initialized with [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT](#) or [WDF\\_OBJECT\\_ATTRIBUTES\\_INIT\\_CONTEXT\\_TYPE](#). This parameter is optional and can be [WDF\\_NO\\_OBJECT\\_ATTRIBUTES](#).

[out] UfxEndpoint

A pointer to a location that receives a handle to a UFXENDPOINT object.

## Return value

If the operation is successful, the method returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The transfer queue handles the following IOCTLs related to endpoint transfers:

- [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_IN](#)
- [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_OUT](#)
- [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN](#)
- [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN\\_APPEND\\_ZERO\\_PKT](#)
- [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_OUT](#)

The command queue will handle the following IOCTLs:

- [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_PIPE\\_STATE](#)
- [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_PIPE\\_STATE](#)

- [IOCTL\\_INTERNAL\\_USBFN\\_DESCRIPTOR\\_UPDATE](#)

The following example shows how to create a UFXENDPOINT object and initialize its context.

```
NTSTATUS
UfxEndpointAdd (
    _In_ UFXDEVICE Device,
    _In_ PUSB_ENDPOINT_DESCRIPTOR Descriptor,
    _Inout_ PUFXENDPOINT_INIT EndpointInit
)
/*++
Routine Description:

    Creates a UFXENDPOINT object, and initializes its contexts.

Parameters Description:

    Device - UFXDEVICE associated with the endpoint.

    Descriptor - Endpoint descriptor for this endpoint.

    EndpointInit - Opaque structure from UFX.

Return Value:

    STATUS_SUCCESS if successful, appropriate NTSTATUS message otherwise.

--*/
{
    NTSTATUS Status;
    WDF_OBJECT_ATTRIBUTES Attributes;
    WDF_IO_QUEUE_CONFIG TransferQueueConfig;
    WDF_OBJECT_ATTRIBUTES TransferQueueAttributes;
    WDF_IO_QUEUE_CONFIG CommandQueueConfig;
    WDF_OBJECT_ATTRIBUTES CommandQueueAttributes;
    UFXENDPOINT Endpoint;
    PUFXENDPOINT_CONTEXT EpContext;
    PUFXDEVICE_CONTEXT DeviceContext;
    UFX_ENDPOINT_CALLBACKS Callbacks;
    PENDPOINT_QUEUE_CONTEXT QueueContext;
    WDFQUEUE Queue;

    TraceEntry();

    DeviceContext = UfxDeviceGetContext(Device);

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&Attributes,
    UFXENDPOINT_CONTEXT);
    Attributes.ExecutionLevel = WdfExecutionLevelPassive;
    Attributes.EvtCleanupCallback = UfxEndpoint_Cleanup;
```

```

// Note: Execution level needs to be passive to avoid deadlocks with
WdfRequestComplete.
//
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&TransferQueueAttributes,
ENDPOINT_QUEUE_CONTEXT);
TransferQueueAttributes.ExecutionLevel = WdfExecutionLevelPassive;

WDF_IO_QUEUE_CONFIG_INIT(&TransferQueueConfig,
WdfIoQueueDispatchManual);
TransferQueueConfig.AllowZeroLengthRequests = TRUE;
TransferQueueConfig.EvtIoStop = EndpointQueue_EvtIoStop;

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&CommandQueueAttributes,
ENDPOINT_QUEUE_CONTEXT);
CommandQueueAttributes.ExecutionLevel = WdfExecutionLevelPassive;

WDF_IO_QUEUE_CONFIG_INIT(&CommandQueueConfig,
WdfIoQueueDispatchSequential);
CommandQueueConfig.EvtIoInternalDeviceControl = EvtEndpointCommandQueue;

UFX_ENDPOINT_CALLBACKS_INIT(&Callbacks);
UfxEndpointInitSetEventCallbacks(EndpointInit, &Callbacks);

Status = UfxEndpointCreate(
    Device,
    EndpointInit,
    &Attributes,
    &TransferQueueConfig,
    &TransferQueueAttributes,
    &CommandQueueConfig,
    &CommandQueueAttributes,
    &Endpoint);
CHK_NT_MSG(Status, "Failed to create ufxendpoint!");

Status = WdfCollectionAdd(DeviceContext->Endpoints, Endpoint);
CHK_NT_MSG(Status, "Failed to add endpoint to collection!");

EpContext = UfxEndpointGetContext(Endpoint);
EpContext->UfxDevice = Device;
EpContext->WdfDevice = DeviceContext->FdoWdfDevice;
RtlCopyMemory(&EpContext->Descriptor, Descriptor, sizeof(*Descriptor));

Queue = UfxEndpointGetTransferQueue(Endpoint);
QueueContext = EndpointQueueGetContext(Queue);
QueueContext->Endpoint = Endpoint;

Queue = UfxEndpointGetCommandQueue(Endpoint);
QueueContext = EndpointQueueGetContext(Queue);
QueueContext->Endpoint = Endpoint;

Status = TransferInitialize(Endpoint);
CHK_NT_MSG(Status, "Failed to initialize endpoint transfers");

//

```

```

// This can happen if we're handling a SetInterface command.
//
if (DeviceContext->UsbState == UsbfnDeviceStateConfigured) {
    UfxEndpointConfigure(Endpoint);
}

Status = WdfIoQueueReadyNotify(
    UfxEndpointGetTransferQueue(Endpoint),
    TransferReadyNotify,
    Endpoint);
CHK_NT_MSG(Status, "Failed to register ready notify");

End:
    TraceExit();
    return Status;
}

```

## Requirements

Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	PASSIVE_LEVEL

# UfxEndpointGetCommandQueue function (ufxclient.h)

Article 02/22/2024

Returns the command queue previously created by [UfxEndpointCreate](#).

## Syntax

C++

```
WDFQUEUE UfxEndpointGetCommandQueue(
    [in] UFXENDPOINT UfxEndpoint
);
```

## Parameters

[in] `UfxEndpoint`

A handle to an endpoint object returned from a previous call to [UfxEndpointCreate](#).

## Return value

A handle to a framework queue object.

## Remarks

For an code example that shows how to create an endpoint object and initialize its context, see the Remarks section of [UfxEndpointCreate](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows

Requirement	Value
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

## See also

[UfxEndpointCreate](#)

# UfxEndpointGetTransferQueue function (ufxclient.h)

Article 02/22/2024

Returns the transfer queue previously created by [UfxEndpointCreate](#).

## Syntax

C++

```
WDFQUEUE UfxEndpointGetTransferQueue(
    [in] UFXENDPOINT UfxEndpoint
);
```

## Parameters

[in] `UfxEndpoint`

A handle to an endpoint object returned from a previous call to [UfxEndpointCreate](#).

## Return value

A handle to a framework queue object.

## Remarks

For an code example that shows how to create an endpoint object and initialize its context, see the Remarks section of [UfxEndpointCreate](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows

Requirement	Value
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

## See also

[UfxEndpointCreate](#)

# UfxEndpointInitSetEventCallbacks function (ufxclient.h)

Article 02/22/2024

Initialize a **UFXENDPOINT\_INIT** structure.

## Syntax

C++

```
void UfxEndpointInitSetEventCallbacks(
    [in, out] PUFXENDPOINT_INIT          EndpointInit,
    [in]      PUFX_ENDPOINT_CALLBACKS  Callbacks
);
```

## Parameters

[in, out] **EndpointInit**

Opaque structure passed by UFX in the call to [EVT\\_UFX\\_DEVICE\\_ENDPOINT\\_ADD](#) or [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#).

[in] **Callbacks**

Pointer to a [UFX\\_ENDPOINT\\_CALLBACKS](#) structure.

## Return value

None

## Remarks

The client driver calls [UfxEndpointCreate](#) from its [EVT\\_UFX\\_DEVICE\\_ENDPOINT\\_ADD](#) or [EVT\\_UFX\\_DEVICE\\_DEFAULT\\_ENDPOINT\\_ADD](#) event callback function in order to create a new endpoint.

The client driver first calls [UFX\\_ENDPOINT\\_CALLBACKS\\_INIT](#) to initialize a [UFX\\_ENDPOINT\\_CALLBACKS](#) structure. Then it calls [UfxEndpointCreate](#) with the initialized [UFX\\_ENDPOINT\\_CALLBACKS](#) structure.

For an code example that shows how to create a UFXENDPOINT object and initialize its context, see the Remarks section of [UfxEndpointCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	PASSIVE_LEVEL

# UfxEndpointNotifySetup function (ufxclient.h)

Article02/22/2024

Notifies UFX when the client driver receives a setup packet from the host.

## Syntax

C++

```
void UfxEndpointNotifySetup(
    UFXENDPOINT                 UfxEndpoint,
    [in] PUSB_DEFAULT_PIPE_SETUP_PACKET SetupInfo
);
```

## Parameters

UfxEndpoint

A handle to a UFX device object that the driver created by calling [UfxDeviceCreate](#).

[in] SetupInfo

A pointer to a USB setup packet described in a **USB\_DEFAULT\_PIPE\_SETUP\_PACKET** structure (defined in Usbspec.h).

## Return value

None

## Remarks

The following example shows how to handle setup packet completion.

```
if (ControlContext->SetupRequested) {
    TRACE_TRANSFER("COMPLETE (Setup)", Endpoint, NULL);

    ControlContext->SetupRequested = FALSE;
    TransferContext->TransferStarted = FALSE;
```

```
    UfxEndpointNotifySetup(Endpoint, ControlContext->SetupPacket);

}
```

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	DISPATCH_LEVEL

# UfxFdoInit function (ufxclient.h)

Article02/22/2024

Initializes the WDFDEVICE\_INIT structure that the client driver subsequently provides when it calls [WdfDeviceCreate](#).

## Syntax

C++

```
NTSTATUS UfxFdoInit(
    [in]      WDFDRIVER             WdfDriver,
    [in, out] PWDFDEVICE_INIT       DeviceInit,
    [in, out] PWDF_OBJECT_ATTRIBUTES FdoAttributes
);
```

## Parameters

[in] `WdfDriver`

A handle to the driver's WDF driver object that the driver obtained from a previous call to [WdfDriverCreate](#) or [WdfGetDriver](#).

[in, out] `DeviceInit`

A pointer to a [WDFDEVICE\\_INIT](#) structure.

[in, out] `FdoAttributes`

A pointer to a caller-allocated [WDF\\_OBJECT\\_ATTRIBUTES](#) structure that describes object attributes for the

## Return value

If the operation is successful, the method returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The client driver receives a pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure in its [EvtDriverDeviceAdd](#) callback function. It then calls [UfxFdoInit](#) with this pointer before calling [WdfDeviceCreate](#) to create the WDFDEVICE object.

By default, for WDF drivers, the device's function driver is the power policy owner.

The following code snippet shows how to call [UfxFdoInit](#).

```
NTSTATUS
UfxClientDeviceCreate(
    _In_ WDFDRIVER Driver,
    _In_ PWDDEVICE_INIT DeviceInit
)
/*++

Routine Description:

    Worker routine called to create a device and its software resources.

Arguments:

    Driver - WDF driver object

    DeviceInit - Pointer to an opaque init structure. Memory for this
                 structure will be freed by the framework when
WdfDeviceCreate
                 succeeds. So don't access the structure after that point.

Return Value:

    Appropriate NTSTATUS value

--*/
{
    WDF_OBJECT_ATTRIBUTES DeviceAttributes;
    WDFDEVICE WdfDevice;

    PAGED_CODE();

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&DeviceAttributes,
CONTROLLER_CONTEXT);

    //
    // Do UFX-specific initialization
    //
    Status = UfxFdoInit(Driver, DeviceInit, &DeviceAttributes);

    //
    // Proceed to WdfDeviceCreate
    //
}
```

}

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10
Target Platform	Windows
Header	ufxclient.h
Library	ufxstub.lib
IRQL	PASSIVE_LEVEL

# ufxproprietarycharger.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

ufxproprietarycharger.h contains the following programming interfaces:

## Callback functions

### [UFX\\_PROPRIETARY\\_CHARGER\\_ABORT\\_OPERATION](#)

The filter driver's implementation to abort a charger operation.

### [UFX\\_PROPRIETARY\\_CHARGER\\_DETECT](#)

The filter driver's implementation to detect if a charger is attached and get details about the charger.

### [UFX\\_PROPRIETARY\\_CHARGER\\_RESET\\_OPERATION](#)

The filter driver's implementation to reset a charger operation.

### [UFX\\_PROPRIETARY\\_CHARGER\\_SET\\_PROPERTY](#)

The filter driver's implementation to set a configurable property on the charger.

## Structures

### [UFX\\_INTERFACE\\_PROPRIETARY\\_CHARGER](#)

Stores pointers to driver-implemented callback functions for handling proprietary charger operations.

### [UFX\\_PROPRIETARY\\_CHARGER](#)

Describes the proprietary charger's device power requirements.

# UFX\_INTERFACE\_PROPRIETARY\_CHARGE R structure (ufxproprietarycharger.h)

Article 02/22/2024

Stores pointers to driver-implemented callback functions for handling proprietary charger operations.

## Syntax

C++

```
typedef struct _UFX_INTERFACE_PROPRIETARY_CHARGER {
    INTERFACE                                InterfaceHeader;
    PFN_UFX_PROPRIETARY_CHARGER_DETECT        ProprietaryChargerDetect;
    PFN_UFX_PROPRIETARY_CHARGER_SET_PROPERTY   ProprietaryChargerSetProperty;
    PFN_UFX_PROPRIETARY_CHARGER_ABORT_OPERATION ProprietaryChargerAbortOperation;
    PFN_UFX_PROPRIETARY_CHARGER_RESET_OPERATION ProprietaryChargerResetOperation;
} UFX_INTERFACE_PROPRIETARY_CHARGER, *PUFX_INTERFACE_PROPRIETARY_CHARGER;
```

## Members

InterfaceHeader

The interface version number.

ProprietaryChargerDetect

A pointer to the driver's implementation of the [UFX\\_PROPRIETARY\\_CHARGER\\_DETECT](#) callback function.

ProprietaryChargerSetProperty

A pointer to the driver's implementation of the [UFX\\_PROPRIETARY\\_CHARGER\\_SET\\_PROPERTY](#) callback function.

ProprietaryChargerAbortOperation

A pointer to the driver's implementation of the [UFX\\_PROPRIETARY\\_CHARGER\\_ABORT\\_OPERATION](#) callback function.

## ProprietaryChargerResetOperation

A pointer to the driver's implementation of the [UFX\\_PROPRIETARY\\_CHARGER\\_RESET\\_OPERATION](#) callback function.

# Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	ufxproprietarycharger.h (include Ufxproprietarycharger.h)

## See also

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

# UFX\_PROPRIETARY\_CHARGER structure (ufxproprietarycharger.h)

Article02/22/2024

Describes the proprietary charger's device power requirements.

## Syntax

C++

```
typedef struct _UFX_PROPRIETARY_CHARGER {
    GUID           ChargerId;
    DEVICE_POWER_STATE DxState;
} UFX_PROPRIETARY_CHARGER, *PUFX_PROPRIETARY_CHARGER;
```

## Members

ChargerId

Charger identifier used to identify a specific type of charger.

DxState

The minimum required device power state when it is connected, indicated by one of the [DEVICE\\_POWER\\_STATE](#)-typed flags.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	ufxproprietarycharger.h

# UFX\_PROPRIETARY\_CHARGER\_ABORT\_OPERATION callback function (ufxproprietarycharger.h)

Article02/22/2024

The filter driver's implementation to abort a charger operation.

## Syntax

C++

```
UFX_PROPRIETARY_CHARGER_ABORT_OPERATION UfxProprietaryChargerAbortOperation;

NTSTATUS UfxProprietaryChargerAbortOperation(
    [in] PVOID Context
)
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support handling of proprietary chargers, the USB lower filter driver must publish support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

# Requirements

 Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxproprietarycharger.h
IRQL	<=DISPATCH_LEVEL

## See also

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

# UFX\_PROPRIETARY\_CHARGER\_DETECT callback function (ufxproprietarycharger.h)

Article02/22/2024

The filter driver's implementation to detect if a charger is attached and get details about the charger.

## Syntax

C++

```
UFX_PROPRIETARY_CHARGER_DETECT UfxProprietaryChargerDetect;  
  
NTSTATUS UfxProprietaryChargerDetect(  
    [in] PVOID Context,  
    [out] PUFX_PROPRIETARY_CHARGER DetectedCharger  
)  
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

[out] DetectedCharger

A pointer to a [UFX\\_PROPRIETARY\\_CHARGER](#) structure that the driver fills with charger information.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support handling of proprietary chargers, the USB lower filter driver must publish support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

In this callback function, the driver assigns the charger a GUID and sets the minimum required Dx state when the device is connected for charging.

## Examples

```
NTSTATUS
UsbLowerFilter_ProprietaryChargerDetect(
    __in PVOID Context,
    __out PUFX_PROPRIETARY_CHARGER DetectedCharger
)
{
    NTSTATUS Status = STATUS_SUCCESS;

    PPDPCP_CONTEXT PdcpContext = NULL;

    PAGED_CODE();

    PdcpContext = DeviceGetUsbLowerFilterContext((WDFDEVICE)Context);

    // Clear our event
    KeClearEvent(&PdcpContext>AbortOperation);

    // Wait for a while
    Timeout.QuadPart =
        WDF_REL_TIMEOUT_IN_MS(PdcpContext>DetectionDelayInms);

    Status = KeWaitForSingleObject(
        &PdcpContext>AbortOperation,
        Executive,
        KernelMode,
        FALSE,
        &Timeout);

    switch (Status)
    {
        case STATUS_SUCCESS:

            // The abort event was set. Abort.

            Status = STATUS_REQUEST_ABORTED;
            break;
    }
}
```

```

case STATUS_TIMEOUT:

    // Timed out, detection has completed successfully.
    // Check if we want to fail this.

    if (PdcpContext->RejectNextRequest)
    {
        PdcpContext->RejectNextRequest = FALSE;
        Status = STATUS_UNSUCCESSFUL;
    }
    else if (!PdcpContext->PdcpChargerAttached)
    {
        Status = STATUS_NOT_FOUND;
    }
    else
    {
        Status = STATUS_SUCCESS;
    }
    break;

default:
    break;
}

if (NT_SUCCESS(Status))
{
    PdcpContext->PdcpChargerDetected = TRUE;
    DetectedCharger->ChargerId = GUID_USBPN_PROPRIETARY_CHARGER;
    DetectedCharger->DxState = PowerDeviceD2;
}

return Status;
}

```

## Requirements

[Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxproprietarycharger.h
IRQL	PASSIVE_LEVEL

## See also

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

# UFX\_PROPRIETARY\_CHARGER\_RESET\_OPERATION callback function (ufxproprietarycharger.h)

Article 10/21/2021

The filter driver's implementation to reset a charger operation.

## Syntax

C++

```
UFX_PROPRIETARY_CHARGER_RESET_OPERATION UfxProprietaryChargerResetOperation;

NTSTATUS UfxProprietaryChargerResetOperation(
    [in] PVOID Context
)
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support handling of proprietary chargers, the USB lower filter driver must publish support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

# Requirements

Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxproprietarycharger.h
IRQL	<=DISPATCH_LEVEL

## See also

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

# UFX\_PROPRIETARY\_CHARGER\_SET\_PROPERTY callback function (ufxproprietarycharger.h)

Article02/22/2024

The filter driver's implementation to set a configurable property on the charger.

## Syntax

C++

```
UFX_PROPRIETARY_CHARGER_SET_PROPERTY UfxProprietaryChargerSetProperty;

NTSTATUS UfxProprietaryChargerSetProperty(
    [in] PVOID Context,
    [in] PCONFIGURABLE_CHARGER_PROPERTY_HEADER Property
)
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

[in] Property

A pointer to a **CONFIGURABLE\_CHARGER\_PROPERTY\_HEADER** structure (defined in charging.h) that describes the configurable charger property to set.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support handling of proprietary chargers, the USB lower filter driver must publish support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

In this callback function, the driver sets the specified property value. For example, after the detection of a HVDCP charger, the driver sets the output voltages to the specified value.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	ufxproprietarycharger.h
IRQL	PASSIVE_LEVEL

## See also

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

# ursdevice.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

ursdevice.h contains the following programming interfaces:

## Functions

<a href="#">URS_CONFIG_INIT</a>
Initializes a URS_CONFIG structure.
<a href="#">UrsDeviceInitialize</a>
Initializes a framework device object to support operations related to a USB dual-role controller and registers the relevant event callback functions with the USB dual-role controller class extension.
<a href="#">UrsDeviceInitInitialize</a>
Learn how this function initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.
<a href="#">UrsIoResourceListAppendDescriptor</a>
Appends the specified resource descriptor to the specified I/O resource list object that maintains resource descriptors for the host or function role.
<a href="#">UrsReportHardwareEvent</a>
Notifies the USB dual-role class extension about a new hardware event.
<a href="#">UrsSetHardwareEventSupport</a>
Indicates the client driver's support for reporting new hardware events.
<a href="#">UrsSetPoHandle</a>
Registers and deletes the client driver's registration with the power management framework (PoFx).

# Callback functions

## [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#)

The USB dual-role class extension invokes this callback to allow the client driver to insert the resources from the resource-requirements-list object to resource lists that will be used during the life time of each role.

## [EVT\\_URS\\_SET\\_ROLE](#)

The URS class extension invokes this event callback when it requires the client driver to change the role of the controller.

# Structures

## [URS\\_CONFIG](#)

Contains pointers to event callback functions implemented by the URS client driver for a USB dual-role controller. Initialize this structure by calling [URS\\_CONFIG\\_INIT](#).

# EVT\_URS\_DEVICE\_FILTER\_RESOURCE\_REQUIREMENTS callback function (ursdevice.h)

Article 10/21/2021

The USB dual-role class extension invokes this callback to allow the client driver to insert the resources from the resource-requirements-list object to resource lists that will be used during the life time of each role.

## Syntax

C++

```
EVT_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS
EvtUrsDeviceFilterResourceRequirements;

NTSTATUS EvtUrsDeviceFilterResourceRequirements(
    [in] WDFDEVICE Device,
    [in] WDFIORESREQLIST IoResourceRequirementsList,
    [in] URSIORESLIST HostRoleResources,
    [in] URSIORESLIST FunctionRoleResources
)
{...}
```

## Parameters

[in] `Device`

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] `IoResourceRequirementsList`

A handle to a framework resource-requirements-list object that represents a device's resource requirements list.

[in] `HostRoleResources`

A handle to a resource list for the controller device when it is operating in host mode.

[in] `FunctionRoleResources`

A handle to a resource list for the controller when it is operating in function mode.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

The client driver registers its implementation with the USB dual-role class extension by calling [UrsDeviceInitialize](#) after calling [WdfDeviceCreate](#) to create the framework device object for the controller. The class extension invokes this callback before [EvtDevicePrepareHardware](#). The callback is invoked within the class extension's [EvtDeviceFilterRemoveResourceRequirements](#), which is registered on behalf of the client driver. The client must not implement and register its [EvtDeviceFilterRemoveResourceRequirements](#) because it will override the class extension's implementation.

Each role has a certain number of assigned hardware resources. Those resources can be memory, interrupts, and so on. The resources are maintained by the system in a *resource requirements list* that contains the range of hardware resources in which the device can operate.

For more information about resource requirements lists, see [Handling Hardware Resources](#).

The class extension allocates memory for the *resource requirements list* and *resource lists* for both host and function roles. When the class extension invokes the client driver's implementation of *EVT\_URS\_DEVICE\_FILTER\_RESOURCE\_REQUIREMENTS*, it passes a WDFIORESREQLIST handle to that requirements list along with URSIORESLIST handles for host and function role *resource lists*. In the implementation, the client driver is expected to enumerate through the logical configurations in the requirements list and check the resource descriptor for each configuration by calling [WdfIoResourceListGetDescriptor](#).

If the driver wants to use a particular resource, it can add the associated resource descriptor to the respective resource list by calling [UrsIoResourceListAppendDescriptor](#).

To delete a resource descriptor from the requirement list, the driver calls [WdfIoResourceListRemove](#).

## Examples

```
EVT_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS
EvtUrsFilterRemoveResourceRequirements;

_Function_class_(EVT_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS)
_IRQL_requires_same_
_IRQL_requires_max_(PASSIVE_LEVEL)
NTSTATUS
EvtUrsFilterRemoveResourceRequirements (
    _In_ WDFDEVICE Device,
    _In_ WDFIORESREQLIST IoResourceRequirementsList,
    _In_ URSIORESLIST HostRoleResources,
    _In_ URSIORESLIST FunctionRoleResources
)
{

    NTSTATUS status;
    WDFIORESLIST resList;
    ULONG resListCount;
    ULONG resCount;
    ULONG currentResourceIndex;
    PIO_RESOURCE_DESCRIPTOR descriptor;
    BOOLEAN assignToHost;
    BOOLEAN assignToFunction;
    BOOLEAN keepAssigned;

    TRY {

        status = STATUS_SUCCESS;

        //
        // Currently does not support multiple logical configurations. Only
        the first one
        // is considered.
        //

        resListCount =
WdfIoResourceRequirementsListGetCount(IoResourceRequirementsList);
        if (resListCount == 0) {
            // No logical resource configurations found.
            LEAVE;
        }

        // Enumerate through logical resource configurations.

        resList =
WdfIoResourceRequirementsListGetIoResList(IoResourceRequirementsList, 0);
        resCount = WdfIoResourceListGetCount(resList);
```

```

        for (currentResourceIndex = 0; currentResourceIndex < resCount;
++currentResourceIndex) {

            descriptor = WdfIoResourceListGetDescriptor(resList,
currentResourceIndex);

            if (descriptor->Type == CmResourceTypeConfigData) {

                //
                // This indicates the priority of this logical
configuration.
                // This descriptor can be ignored.
                //

                keepAssigned = TRUE;
                assignToFunction = FALSE;
                assignToHost = FALSE;

            } else if ((descriptor->Type == CmResourceTypeMemory) ||
(descriptor->Type == CmResourceTypeMemoryLarge)) {

                //
                // This example client driver keeps the memory resources
here.
                //

                keepAssigned = TRUE;
                assignToFunction = TRUE;
                assignToHost = TRUE;

            } else {

                //
                // For all other resources, pass it to the child device
nodes for host and function.
                //

                keepAssigned = FALSE;
                assignToHost = TRUE;
                assignToFunction = TRUE;
            }

            if (assignToHost != FALSE) {
                status =
UrsIoResourceListAppendDescriptor(HostRoleResources, descriptor);
                if (!NT_SUCCESS(status)) {
                    // UrsIoResourceListAppendDescriptor for
HostRoleResources failed.
                    LEAVE;
                }
            }

            if (assignToFunction != FALSE) {
                status =
UrsIoResourceListAppendDescriptor(FunctionRoleResources, descriptor);
            }
        }
    }
}

```

```

        if (!NT_SUCCESS(status)) {
            // UrsIoResourceListAppendDescriptor for
            FunctionRoleResources failed.
            LEAVE;
        }
    }

    if (keepAssigned == FALSE) {
        WdfIoResourceListRemove(resList, currentResourceIndex);
        --currentResourceIndex;
        --resCount;
    }
}

} FINALLY {
}

return status;
}

```

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include UrsCx.h)
IRQL	PASSIVE_LEVEL

## See also

[Handling Hardware Resources](#)

[UrsDeviceInitialize](#)

[UrsIoResourceListAppendDescriptor](#)

[WdfIoResourceListRemove](#)

# EVT\_URS\_SET\_ROLE callback function (ursdevice.h)

Article02/22/2024

The URS class extension invokes this event callback when it requires the client driver to change the role of the controller.

## Syntax

C++

```
EVT_URS_SET_ROLE EvtUrsSetRole;

NTSTATUS EvtUrsSetRole(
    [in] WDFDEVICE Device,
    [in] URS_ROLE Role
)
{...}
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] Role

A [URS\\_ROLE](#) type value that indicates the role to set for the controller device.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To register the client driver's implementation of the event callback the driver must set the [EvtUrsSetRole](#) member of [URS\\_CONFIG](#) to a function pointer of the implementation

method and then call the [UrsDeviceInitialize](#) method by passing the populated structure. The driver must call the method after it creates the framework device object for the controller.

## Examples

```
NTSTATUS
EvtUrsSetRole (
    _In_ WDFDEVICE Device,
    _In_ URS_ROLE Role
)
{
    NTSTATUS status;
    PFDO_CONTEXT fdoContext;

    TRACE_FUNC_ENTRY(TRACE_FLAG);
    TRY {

        // Change the current role of the controller to the specified
        role.
        // The driver might have stored the control registers in the
        device context.
        // Read and write the register to get and set the current role.

    }

    TRACE_INFO(TRACE_FLAG, "[Device: 0x%p] Successfully set role to
    %!URS_ROLE!", Device, Role);

    status = STATUS_SUCCESS;

} FINALLY {

}

TRACE_FUNC_EXIT(TRACE_FLAG);

return status;
}
```

## Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include Urscx.h)
IRQL	PASSIVE_LEVEL

## See also

[UrsDeviceInitialize](#)

# URS\_CONFIG structure (ursdevice.h)

Article02/22/2024

Contains pointers to event callback functions implemented by the URS client driver for a USB dual-role controller. Initialize this structure by calling [URS\\_CONFIG\\_INIT](#).

## Syntax

C++

```
typedef struct _URS_CONFIG {
    ULONG                                     Size;
    URS_HOST_INTERFACE_TYPE                   HostInterfaceType;
    PFN_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS
    EvtUrsFilterRemoveResourceRequirements;
    PFN_URS_SET_ROLE                         EvtUrsSetRole;
} URS_CONFIG, *PURS_CONFIG;
```

## Members

Size

The size of this structure.

HostInterfaceType

A [URS\\_HOST\\_INTERFACE\\_TYPE](#) type value that indicates the type of USB host controller: EHCl, xHCl, or other.

EvtUrsFilterRemoveResourceRequirements

A pointer to an [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#) callback function.

EvtUrsSetRole

A pointer to an [EVT\\_URS\\_SET\\_ROLE](#) callback function.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum KMDF version	1.0
Header	ursdevice.h (include Urscx.h)

## See also

[URS\\_CONFIG\\_INIT](#)

[UrsDeviceInitialize](#)

# URS\_CONFIG\_INIT function (ursdevice.h)

Article10/21/2021

Initializes a [URS\\_CONFIG](#) structure.

## Syntax

C++

```
void URS_CONFIG_INIT(
    [out] PURS_CONFIG Config,
    [in] URS_HOST_INTERFACE_TYPE HostInterfaceType,
    [in] PFN_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS
        EvtUrsFilterRemoveResourceRequirements
);
```

## Parameters

[out] Config

A pointer to a [URS\\_CONFIG](#) structure to initialize.

[in] HostInterfaceType

A [URS\\_HOST\\_INTERFACE\\_TYPE](#) type value that indicates the type of host controller that the dual-role controller implements.

[in] EvtUrsFilterRemoveResourceRequirements

A pointer to a [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#) callback function that is implemented by the client driver.

## Return value

None

## Requirements

Target Platform

Windows

Minimum KMDF version	1.0
Header	ursdevice.h (include Urscx.h)

## See also

[URS\\_CONFIG](#)

[UrsDeviceInitialize](#)

# UrsDeviceInitialize function (ursdevice.h)

Article02/22/2024

Initializes a framework device object to support operations related to a USB dual-role controller and registers the relevant event callback functions with the USB dual-role controller class extension.

## Syntax

C++

```
NTSTATUS UrsDeviceInitialize(
    [in] WDFDEVICE Device,
    [in] PURS_CONFIG Config
);
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] Config

A pointer to a [URS\\_CONFIG](#) structure that the client driver initialized by calling [URS\\_CONFIG\\_INIT](#).

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the USB dual-role controller must call this method after the [WdfDeviceCreate](#) call.

The client driver calls this method in its [EvtDriverDeviceAdd](#) implementation.

During this call, the client driver-supplied event callback implementations are also registered by setting appropriate members of [URS\\_CONFIG](#).

The method creates resource lists for host and function roles and the queues required to handle IOCTL requests that are sent to the controller. With each role switch operation, the current role's child device stack is torn down and the device stack for the new role is loaded. The [UrsDeviceInitialize](#) method retrieves identifying information that is used to build those device stacks. The method also retrieves information about the device from the underlying bus, such as ACPI.

## Examples

```
EVT_URS_DEVICE_FILTER_RESOURCE_REQUIREMENTS
EvtUrsFilterResourceRequirements;
EVT_URS_SET_ROLE EvtUrsSetRole;

EvtDriverDeviceAdd (
    _In_ WDFDRIVER Driver,
    _Inout_ PWDFDEVICE_INIT DeviceInit
)
{
...
    WDFDEVICE device;
    NTSTATUS status;
    WDF_OBJECT_ATTRIBUTES attributes;
    URS_CONFIG ursConfig;
...
    status = UrsDeviceInitInitialize(DeviceInit);
    if (!NT_SUCCESS(status)) {
        //UrsDeviceInitInitialize failed.
        return status;
    }

    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, DRIVER_CONTEXT);
    status = WdfDeviceCreate(&DeviceInit, &attributes, &device);
    if (!NT_SUCCESS(status)) {
        // WdfDeviceCreate failed.
        return status;
    }
    URS_CONFIG_INIT(&ursConfig, UrsHostInterfaceTypeXhci,
    EvtUrsFilterResourceRequirements);
```

```

ursConfig.EvtUrsSetRole = EvtUrsSetRole;
status = UrsDeviceInitialize(device, &ursConfig);

if (!NT_SUCCESS(status)) {
    // UrsDeviceInitialize failed.
    return status;

}

...
}

```

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include Urscx.h)
Library	Urscxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[URS\\_CONFIG](#)

[URS\\_CONFIG\\_INIT](#)

# UrsDeviceInitInitialize function (ursdevice.h)

Article 02/22/2024

Initializes device initialization operations when the Plug and Play (PnP) manager reports the existence of a device.

## Syntax

C++

```
NTSTATUS UrsDeviceInitInitialize(
    PWDFDEVICE_INIT DeviceInit
);
```

## Parameters

DeviceInit

A pointer to a framework-allocated [WDFDEVICE\\_INIT](#) structure.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

The client driver for the dual-role controller calls this method in its [EvtDriverDeviceAdd](#) implementation before it calls [WdfDeviceCreate](#) and [UrsDeviceInitialize](#). For code example, see [UrsDeviceInitialize](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include Urscx.h)
Library	UrsCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[WdfDeviceCreate](#)

# UrsIoResourceListAppendDescriptor function (ursdevice.h)

Article02/22/2024

Appends the specified resource descriptor to the specified I/O resource list object that maintains resource descriptors for the host or function role.

## Syntax

C++

```
NTSTATUS UrsIoResourceListAppendDescriptor(
    [in] URSIORESLIST          IoResourceList,
    [in] PIO_RESOURCE_DESCRIPTOR Descriptor
);
```

## Parameters

[in] IoResourceList

A role's I/O resource list object to which the resource descriptor is appended. This object is allocated by the framework and passed to the client driver when the framework invokes the driver's [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#) implementation.

[in] Descriptor

A pointer to IO\_RESOURCE\_DESCRIPTOR that contains the resource descriptor for the role.

## Return value

The method returns STATUS\_SUCCESS if the operation succeeds. Otherwise, this method might return an appropriate [NTSTATUS](#) error code.

## Remarks

After the client driver calls [UrsDeviceInitialize](#), the framework allocates memory for the *resource requirements list*. When the USB dual-role class extension invokes the client

driver's implementation of [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#), it passes a WDFIORESREQLIST handle to that requirements list along with URSIORESLIST handles for host and function role *resource lists*. In the implementation, the client driver is expected to enumerate through the requirements list and add the resource descriptor (if it wants to use that resource) to the resource list for each role.

To add a resource descriptors for a role, the driver calls [UrsIoResourceListAppendDescriptor](#) and specifies the descriptor and the resource list to which the resource must be added.

For a code example, see [EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#).

For more information about resource requirements lists, see [Handling Hardware Resources](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include UrsCx.h)
Library	UrsCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[EVT\\_URS\\_DEVICE\\_FILTER\\_RESOURCE\\_REQUIREMENTS](#)

[UrsDeviceInitialize](#)

# UrsReportHardwareEvent function (ursdevice.h)

Article02/22/2024

Notifies the USB dual-role class extension about a new hardware event.

## Syntax

C++

```
void UrsReportHardwareEvent(
    [in] WDFDEVICE           Device,
    [in] URS_HARDWARE_EVENT HardwareEvent
);
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] HardwareEvent

A [URS\\_HARDWARE\\_EVENT](#)-type value that indicates the type of event that occurred.

## Return value

None

## Remarks

Before reporting any hardware events, the client driver for the dual-role controller must indicate to the class extension that the driver supports hardware events by calling [UrsSetHardwareEventSupport](#).

The client driver cannot pass [UrsHardwareEventNone](#) as the *HardwareEvent* parameter value. That value is reserved for internal use.

The client driver must call this method to report any hardware event, such as ID-pin interrupts. Typically, in the driver's implementation of the [EvtInterruptlsr](#) callback, the driver reads the ID-pin state and reports the event to the class extension by calling this method.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include Urscx.h)
Library	UrsCxStub.lib
IRQL	HIGH_LEVEL

## See also

- [WdfDeviceCreate](#)
- [URS\\_HARDWARE\\_EVENT](#)
- [UrsSetHardwareEventSupport](#)
- [EvtInterruptlsr](#)

# **UrsSetHardwareEventSupport function (ursdevice.h)**

Article02/22/2024

Indicates the client driver's support for reporting new hardware events.

## Syntax

C++

```
void UrsSetHardwareEventSupport(
    [in] WDFDEVICE Device,
    [in] BOOLEAN    HardwareEventReportingSupported
);
```

## Parameters

[in] `Device`

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] `HardwareEventReportingSupported`

A boolean value that indicates support for reporting hardware events.

TRUE indicates the client driver will report hardware events by calling [UrsReportHardwareEvent](#).

FALSE indicates hardware event reporting is not handled by the client driver.

## Return value

None

## Remarks

Before the client driver can report hardware events, the client driver for the dual-role controller must indicate to the class extension that the driver supports hardware events by calling this method. Typically, the driver calls [UrsSetHardwareEventSupport](#) in the

driver's [EvtDevicePrepareHardware](#) callback function. The driver must not call this method after *EvtDevicePrepareHardware* has returned. Otherwise, the method fails and a break is issued if [Driver Verifier](#) is enabled.

For certain controllers, the client driver might not support role detection before performing a role switch operation. In that case, the client driver must set *HardwareEventReportingSupported* to FALSE. The operating system manages the role of the controller.

Otherwise, if the driver supports role detection, it must set *HardwareEventReportingSupported* to TRUE. This indicates to the class extension that the client driver will handle hardware events, such as ID pin interrupts, and report to the class extension that the role needs to be changed. The driver can report events by calling [UrsReportHardwareEvent](#).

## Examples

```
EVT_WDF_DEVICE_PREPARE_HARDWARE EvtDevicePrepareHardware;

NTSTATUS
EvtDevicePrepareHardware (
    _In_ WDFDEVICE Device,
    _In_ WDFCMRESLIST ResourcesRaw,
    _In_ WDFCMRESLIST ResourcesTranslated
)
{
    ULONG resourceCount;
    BOOLEAN hasHardwareEventSupport;

    UNREFERENCED_PARAMETER(ResourcesRaw);

    TRY {

        resourceCount = WdfCmResourceListGetCount(ResourcesTranslated);

        ...

        // DetermineHardwareEventSupport determines support by inspecting
        resources.
        // Implementation not shown.
        hasHardwareEventSupport =
DetermineHardwareEventSupport(ResourcesRaw);
    }
}
```

```

UrsSetHardwareEventSupport(Device, hasHardwareEventSupport);

    if (hasHardwareEventSupport) {
        UrsReportHardwareEvent(Device, UrsHardwareEventIdGround);
    }

    ...

} FINALLY {

}

return STATUS_SUCCESS;
}

```

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15
Header	ursdevice.h (include UrsCx.h)
Library	UrsCxStub.lib
IRQL	PASSIVE_LEVEL

## See also

[UrsReportHardwareEvent](#)

# UrsSetPoHandle function (ursdevice.h)

Article10/21/2021

Registers and deletes the client driver's registration with the power management framework (PoFx).

## Syntax

C++

```
void UrsSetPoHandle(
    [in] WDFDEVICE Device,
    [in] POHANDLE PoHandle
);
```

## Parameters

[in] Device

A handle to the framework device object that the client driver retrieved in the previous call to [WdfDeviceCreate](#).

[in] PoHandle

A handle that represents the registration of the device with PoFx. The client driver receives this handle from WDF in [EvtDeviceWdmPostPoFxRegisterDevice](#) and [EvtDeviceWdmPrePoFxUnregisterDevice](#) callback functions.

## Return value

None

## Remarks

The client driver for the dual-role controller must be the power policy owner. The driver can receive notifications from the power management framework (PoFx). To do so, after calling [UrsDeviceInitialize](#), the driver must register PoFx callback functions. The client driver registers the device with the power framework directly or obtains a POHANDLE from WDF in [EvtDeviceWdmPostPoFxRegisterDevice](#). After registration is successful, the driver provides that handle to the USB dual-role class extension.

In the client driver's implementation of the [EvtDeviceWdmPostPoFxRegisterDevice](#) callback function, the driver is expected to call [UrsSetPoHandle](#) by passing the received handle. On some platforms, the class extension might use the POHANDLE to power-manage the controller. Conversely, before the class extension deletes the registration with the power framework, it invokes the client driver's [EvtDeviceWdmPrePoFxUnregisterDevice](#) implementation. The driver is expected to call [UrsSetPoHandle](#) by passing NULL as the PoHandle value.

## Examples

```
EVT_WDFDEVICE_WDM_POST_PO_FX_REGISTER_DEVICE EvtDevicePostPoFxRegister;

EVT_WDFDEVICE_WDM_PRE_PO_FX_UNREGISTER_DEVICE EvtDevicePrePoFxUnregister;

EvtDriverDeviceAdd (
    _In_ WDFDRIVER Driver,
    _Inout_ PWDFDEVICE_INIT DeviceInit
)
{
...
...

    WDFDEVICE device;
    NTSTATUS status;
    ...

    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS idleSettings;
    WDF_POWER_FRAMEWORK_SETTINGS poFxSettings;
    ...

    TRY {

        WDF_POWER_FRAMEWORK_SETTINGS_INIT(&poFxSettings);
        poFxSettings.EvtDeviceWdmPostPoFxRegisterDevice =
            EvtDevicePostPoFxRegister;
        poFxSettings.EvtDeviceWdmPrePoFxUnregisterDevice =
            EvtDevicePrePoFxUnregister;

        status = WdfDeviceWdmAssignPowerFrameworkSettings(device,
&poFxSettings);
        if (!NT_SUCCESS(status)) {
            // WdfDeviceWdmAssignPowerFrameworkSettings failed.

        }

        WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT(&idleSettings,
IdleCannotWakeFromS0);
    }
}
```

```

idleSettings.IdleTimeoutType = SystemManagedIdleTimeout;

status = WdfDeviceAssignS0IdleSettings(device, &idleSettings);
if (!NT_SUCCESS(status)) {
    // WdfDeviceAssignS0IdleSettings failed.

}

} FINALLY {

}

..

}

NTSTATUS

EvtDevicePostPoFxRegister (
    _In_ WDFDEVICE Device,
    _In_ POHANDLE PoHandle
)
{
    UrsSetPoHandle(Device, PoHandle);

    return STATUS_SUCCESS;
}

VOID

EvtDevicePrePoFxUnregister (
    _In_ WDFDEVICE Device,
    _In_ POHANDLE PoHandle
)
{
    UNREFERENCED_PARAMETER(PoHandle);

    UrsSetPoHandle(Device, NULL);
}

```

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Target Platform	Windows
Minimum KMDF version	1.15

Header	ursdevice.h (include Urscx.h)
Library	Urscxstub.lib
IRQL	PASSIVE_LEVEL

## See also

[EvtDeviceWdmPostPoFxRegisterDevice](#)

*EvtDeviceWdmPrePoFxUnregisterDevice*

# urstypes.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

urstypes.h contains the following programming interfaces:

## Enumerations

### [URS\\_HARDWARE\\_EVENT](#)

Defines values for the hardware events that a client driver for a USB dual-role controller can report.

### [URS\\_HOST\\_INTERFACE\\_TYPE](#)

Defines values for the various types of USB host controllers.

### [URS\\_ROLE](#)

Defines values for roles supported by a USB dual-role controller.

# URS\_HARDWARE\_EVENT enumeration (urstypes.h)

Article 06/03/2021

Defines values for the hardware events that a client driver for a USB dual-role controller can report.

## Syntax

C++

```
typedef enum _URS_HARDWARE_EVENT {
    UrsHardwareEventNone,
    UrsHardwareEventDetach,
    UrsHardwareEventIdGround,
    UrsHardwareEventIdFloat,
    UrsHardwareEventPortTypeDfp,
    UrsHardwareEventPortTypeUfp
} URS_HARDWARE_EVENT, *PURS_HARDWARE_EVENT;
```

## Constants

`UrsHardwareEventNone`

Internal use only.

`UrsHardwareEventDetach`

A detach event occurred on a port of a USB Type-C system.

`UrsHardwareEventIdGround`

This event indicates that the ID pin is grounded.

`UrsHardwareEventIdFloat`

This event indicates that the ID pin is floating.

`UrsHardwareEventPortTypeDfp`

The Type-C connector has resolved to DFP. Not to be used directly by the URS client driver.

`UrsHardwareEventPortTypeUfp`

The Type-C connector has resolved to UFP. Not to be used directly by the URS client driver.

## Remarks

Values defined for USB Type-C systems should not be directly used by the client driver. Instead the driver should report that it does not support hardware event reporting by calling [UrsSetHardwareEventSupport](#). These hardware events are detected by a USB Type-C connector driver, see [USB Type-C connector driver programming reference](#).

## Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15
Header	urstypes.h (include UrsCx.h)

# URS\_HOST\_INTERFACE\_TYPE enumeration (urstypes.h)

Article02/22/2024

Defines values for the various types of USB host controllers.

## Syntax

C++

```
typedef enum _URS_HOST_INTERFACE_TYPE {
    UrsHostInterfaceTypeEhci,
    UrsHostInterfaceTypeXhci,
    UrsHostInterfaceTypeOther
} URS_HOST_INTERFACE_TYPE;
```

## Constants

[+] Expand table

<code>UrsHostInterfaceTypeEhci</code>	Indicates an EHCI host controller.
<code>UrsHostInterfaceTypeXhci</code>	Indicates an xHCI host controller.
<code>UrsHostInterfaceTypeOther</code>	Indicates a generic host controller.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Requirement	Value
Minimum KMDF version	1.15
Header	urstypes.h (include Urscx.h)

# URS\_ROLE enumeration (urstypes.h)

Article02/22/2024

Defines values for roles supported by a USB dual-role controller.

## Syntax

C++

```
typedef enum _URS_ROLE {
    UrsRoleNone,
    UrsRoleHost,
    UrsRoleFunction
} URS_ROLE, *PURS_ROLE;
```

## Constants

[] Expand table

<code>UrsRoleNone</code>	Internal use only. Must be 0.
<code>UrsRoleHost</code>	Indicates the host role of the controller.
<code>UrsRoleFunction</code>	Indicates the function role of the controller.

## Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Minimum KMDF version	1.15

<b>Requirement</b>	<b>Value</b>
Header	urstypes.h (include Urstypes.h)

# usb.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usb.h contains the following programming interfaces:

## Structures

### [\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#)

The \_URB\_BULK\_OR\_INTERRUPT\_TRANSFER structure is used by USB client drivers to send or receive data on a bulk pipe or on an interrupt pipe.

### [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

The \_URB\_CONTROL\_DESCRIPTOR\_REQUEST structure is used by USB client drivers to get or set descriptors on a USB device.

### [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#)

The \_URB\_CONTROL\_FEATURE\_REQUEST structure is used by USB client drivers to set or clear features on a device, interface, or endpoint.

### [\\_URB\\_CONTROL\\_GET\\_CONFIGURATION\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_CONFIGURATION\_REQUEST structure is used by USB client drivers to retrieve the current configuration for a device.

### [\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_INTERFACE\_REQUEST structure is used by USB client drivers to retrieve the current alternate interface setting for an interface in the current configuration.

### [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#)

The \_URB\_CONTROL\_GET\_STATUS\_REQUEST structure is used by USB client drivers to retrieve status from a device, interface, endpoint, or other device-defined target.

### [\\_URB\\_CONTROL\\_TRANSFER](#)

The \_URB\_CONTROL\_TRANSFER structure is used by USB client drivers to transfer data to or from a control pipe.

## [\\_URB\\_CONTROL\\_TRANSFER\\_EX](#)

The \_URB\_CONTROL\_TRANSFER\_EX structure is used by USB client drivers to transfer data to or from a control pipe, with a timeout that limits the acceptable transfer time.

## [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#)

The \_URB\_CONTROL\_VENDOR\_OR\_CLASS\_REQUEST structure is used by USB client drivers to issue a vendor or class-specific command to a device, interface, endpoint, or other device-defined target.

## [\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#)

The \_URB\_GET\_CURRENT\_FRAME\_NUMBER structure is used by USB client drivers to retrieve the current frame number.

## [\\_URB\\_GET\\_ISOCH\\_PIPE\\_TRANSFER\\_PATH\\_DELAYS](#)

The \_URB\_GET\_ISOCH\_PIPE\_TRANSFER\_PATH\_DELAYS structure is used by USB client drivers to retrieve delays associated with isochronous transfer programming in the host controller and transfer completion so that the client driver can ensure that the device gets the isochronous packets in time.

## [\\_URB\\_HEADER](#)

The \_URB\_HEADER structure is used by USB client drivers to provide basic information about the request being sent to the host controller driver.

## [\\_URB\\_ISOCH\\_TRANSFER](#)

The \_URB\_ISOCH\_TRANSFER structure is used by USB client drivers to send data to or retrieve data from an isochronous transfer pipe.

## [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#)

The \_URB\_OPEN\_STATIC\_STREAMS structure is used by a USB client driver to open streams in the specified bulk endpoint.

## [\\_URB\\_OS\\_FEATURE\\_DESCRIPTOR\\_REQUEST](#)

The \_URB\_OS\_FEATURE\_DESCRIPTOR\_REQUEST structure is used by the USB hub driver to retrieve Microsoft OS Feature Descriptors from a USB device or an interface on a USB device.

## [\\_URB\\_PIPE\\_REQUEST](#)

The \_URB\_PIPE\_REQUEST structure is used by USB client drivers to clear a stall condition on an endpoint.

## [\\_URB\\_SELECT\\_CONFIGURATION](#)

The \_URB\_SELECT\_CONFIGURATION structure is used by client drivers to select a configuration for a USB device.

## [\\_URB\\_SELECT\\_INTERFACE](#)

The \_URB\_SELECT\_INTERFACE structure is used by USB client drivers to select an alternate setting for an interface or to change the maximum packet size of a pipe in the current configuration on a USB device.

## [URB](#)

The URB structure is used by USB client drivers to describe USB request blocks (URBs) that send requests to the USB driver stack. The URB structure defines a format for all possible commands that can be sent to a USB device.

## [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION](#)

Stores xHCI-specific information that is used by client drivers to transfer data to and from the offloaded endpoints.

## [USBD\\_INTERFACE\\_INFORMATION](#)

The USBD\_INTERFACE\_INFORMATION structure holds information about an interface for a configuration on a USB device.

## [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#)

The USBD\_ISO\_PACKET\_DESCRIPTOR structure is used by USB client drivers to describe an isochronous transfer packet.

## [USBD\\_PIPE\\_INFORMATION](#)

The USBD\_PIPE\_INFORMATION structure is used by USB client drivers to hold information about a pipe from a specific interface.

## [USBD\\_STREAM\\_INFORMATION](#)

The USBD\_STREAM\_INFORMATION structure stores information about a stream associated with a bulk endpoint.

## [USBD\\_VERSION\\_INFORMATION](#)

The USBD\_VERSION\_INFORMATION structure is used by the GetUSBDIVersion function to report its output data.

# Enumerations

## [USB\\_CONTROLLER\\_FLAVOR](#)

The USB\_CONTROLLER\_FLAVOR enumeration specifies the type of USB host controller.

## [USBD\\_ENDPOINT\\_OFFLOAD\\_MODE](#)

Defines values for endpoint offloading options in the USB device or host controller.

## [USBD\\_PIPE\\_TYPE](#)

The USBD\_PIPE\_TYPE enumerator indicates the type of pipe.

# \_URB\_BULK\_OR\_INTERRUPT\_TRANSFER structure (usb.h)

Article04/01/2021

The \_URB\_BULK\_OR\_INTERRUPT\_TRANSFER structure is used by USB client drivers to send or receive data on a bulk pipe or on an interrupt pipe.

## Syntax

C++

```
struct _URB_BULK_OR_INTERRUPT_TRANSFER {
    struct _URB_HEADER    Hdr;
    USBD_PIPE_HANDLE      PipeHandle;
    ULONG                 TransferFlags;
    ULONG                 TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                  TransferBufferMDL;
    struct _URB            *UrbLink;
    struct _URB_HCD_AREA  hca;
};
```

## Members

Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

**Hdr.Function** must be URB\_FUNCTION\_BULK\_OR\_INTERRUPT\_TRANSFER, and

**Hdr.Length** must be set to `sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER)`.

PipeHandle

Specifies an opaque handle to the bulk or interrupt pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type URB\_FUNCTION\_SELECT\_CONFIGURATION or when the client driver changes the settings for an interface with a URB of type URB\_FUNCTION\_SELECT\_INTERFACE.

TransferFlags

Specifies zero, one, or a combination of the following flags:

Value	Meaning
-------	---------

<b>USBD_TRANSFER_DIRECTION_IN</b>	Is set to request data from a device. To transfer data to a device, this flag must be clear.
<b>USBD_TRANSFER_DIRECTION_OUT</b>	Is set to transfer data to a device. Setting this flag is equivalent to clearing the <b>USBD_TRANSFER_DIRECTION_IN</b> flag.
<b>USBD_SHORT_TRANSFER_OK</b>	<p>Is set to direct the host controller not to return an error when it receives a packet from the device that is shorter than the maximum packet size for the endpoint. The maximum packet size for the endpoint is reported in the <b>wMaxPacketSize</b> member of the <a href="#">USB_ENDPOINT_DESCRIPTOR</a> structure (endpoint descriptor). When the host controller receives a packet shorter than <b>wMaxPacketSize</b> on a bulk or an interrupt endpoint, the host controller immediately stops requesting data from the endpoint and completes the transfer. If the <b>USBD_SHORT_TRANSFER_OK</b> flag is not set, the host controller completes the transfer with an error.</p> <p>This flag should not be set unless <b>USBD_TRANSFER_DIRECTION_IN</b> is also set. <b>Note</b> On EHCI host controllers, <b>USBD_SHORT_TRANSFER_OK</b> is ignored for bulk and interrupt endpoints. Transfer of short packets on EHCI controllers does not result in an error condition.</p> <p>On UHCI and OHCI host controllers, if <b>USBD_SHORT_TRANSFER_OK</b> 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. The client driver must resume the endpoint by submitting a <a href="#">URB_FUNCTION_SYNC_RESET_PIPE_AND_CLEAR_STALL</a> request before submitting a transfer request to the endpoint.</p>

#### TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified this buffer will contain data read from the

device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### hca

Reserved. Do not use.

## Remarks

Drivers can use the **UsbBuildInterruptOrBulkTransferRequest** service routine to format this URB. Buffers specified in **TransferBuffer** or described in **TransferBufferMDL** must be nonpageable.

In an **URB**, both **TransferBuffer** and **TransferBufferMDL** parameters can be non-**NULL** values, at the same time. In that case, the transfer buffer and the MDL pointed to **TransferBuffer** and **TransferBufferMDL** must point to the same buffer.

The USB bus driver processes this URB at **DISPATCH\_LEVEL**.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

Header	usb.h (include Usb.h)

## See also

URB

USB Structures

\_URB\_HEADER

# \_URB\_CONTROL\_DESCRIPTOR\_REQUEST structure (usb.h)

Article04/01/2021

The \_URB\_CONTROL\_DESCRIPTOR\_REQUEST structure is used by USB client drivers to get or set descriptors on a USB device.

## Syntax

C++

```
struct _URB_CONTROL_DESCRIPTOR_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                Reserved0;
    ULONG                TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                 TransferBufferMDL;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    USHORT                Reserved1;
    UCHAR                 Index;
    UCHAR                 DescriptorType;
    USHORT                LanguageId;
    USHORT                Reserved2;
};
```

## Members

### Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

Hdr.Function must be one of the following:

Hdr.Length must equal `sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST)`.

### Reserved

Reserved. Do not use.

### Reserved0

Reserved. Do not use.

**TransferBufferLength**

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

**TransferBuffer**

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**.

**TransferBufferMDL**

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. This MDL must be allocated from nonpaged pool.

**UrbLink**

Reserved. Do not use.

**hca**

Reserved. Do not use.

**Reserved1**

Reserved. Do not use.

**Index**

Specifies the device-defined index of the descriptor that is being retrieved or set.

**DescriptorType**

Indicates what type of descriptor is being retrieved or set. One of the following values must be specified:

<b>usbspec.h constant</b>	<b>Value</b>	<b>USB Version</b>
USB_DEVICE_DESCRIPTOR_TYPE	0x01	USB 1.1
USB_CONFIGURATION_DESCRIPTOR_TYPE	0x02	USB 1.1
USB_STRING_DESCRIPTOR_TYPE	0x03	USB 1.1
USB_INTERFACE_DESCRIPTOR_TYPE	0x04	USB 1.1

<b>usbspec.h constant</b>	<b>Value</b>	<b>USB Version</b>
USB_ENDPOINT_DESCRIPTOR_TYPE	0x05	USB 1.1
USB_DEVICE_QUALIFIER_DESCRIPTOR_TYPE	0x06	USB 2.0
USB_OTHER_SPEED_CONFIGURATION_DESCRIPTOR_TYPE	0x07	USB 2.0
USB_INTERFACE_POWER_DESCRIPTOR_TYPE	0x08	USB 2.0
USB_OTG_DESCRIPTOR_TYPE	0x09	USB 3.0
USB_DEBUG_DESCRIPTOR_TYPE	0x0a	USB 3.0
USB_INTERFACE_ASSOCIATION_DESCRIPTOR_TYPE	0x0b	USB 3.0
USB_BOS_DESCRIPTOR_TYPE	0x0f	USB 3.0
USB_DEVICE_CAPABILITY_DESCRIPTOR_TYPE	0x10	USB 3.0
USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR_TYPE	0x30	USB 3.0
USB_20_HUB_DESCRIPTOR_TYPE	0x29	USB 1.1 USB 2.0
USB_30_HUB_DESCRIPTOR_TYPE	0x2a	USB 3.0
USB_SUPERSPEEDPLUS_ISOCH_ENDPOINT_COMPANION_DESCRIPTOR_TYPE	0x31	USB 3.1

#### LanguageId

Specifies the language ID of the descriptor to be retrieved when **USB\_STRING\_DESCRIPTOR\_TYPE** is set in **DescriptorType**. This member must be set to zero for any other value in **DescriptorType**.

#### Reserved2

Reserved. Do not use.

## Remarks

Drivers can use the **UsbBuildGetDescriptorRequest** service routine to format this URB. If the caller passes a buffer too small to hold all of the data, the bus driver truncates the data to fit in the buffer without error.

When the caller requests the device descriptor, the bus driver returns a [USB\\_DEVICE\\_DESCRIPTOR](#) data structure.

When the caller requests a configuration descriptor, the bus driver returns the configuration descriptor in a [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure, followed by the interface and endpoint descriptors for that configuration. The driver can access the interface and endpoint descriptors as [USB\\_INTERFACE\\_DESCRIPTOR](#) and [USB\\_ENDPOINT\\_DESCRIPTOR](#) structures. The bus driver also returns any class-specific or device-specific descriptors. The system provides the [USBD\\_ParseConfigurationDescriptorEx](#) and [USBD\\_ParseDescriptors](#) service routines to find individual descriptors within the buffer.

When the caller requests a string descriptor, the bus driver returns a [USB\\_STRING\\_DESCRIPTOR](#) structure. The string itself is found in the variable-length **bString** member of the string descriptor.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[URB](#)

[USB Structures](#)

[USB\\_CONFIGURATION\\_DESCRIPTOR](#)

[USB\\_DEVICE\\_DESCRIPTOR](#)

[USB\\_ENDPOINT\\_DESCRIPTOR](#)

[USB\\_INTERFACE\\_DESCRIPTOR](#)

[USB\\_STRING\\_DESCRIPTOR](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL FEATURE REQUEST structure (usb.h)

Article04/01/2021

The \_URB\_CONTROL FEATURE REQUEST structure is used by USB client drivers to set or clear features on a device, interface, or endpoint.

## Syntax

C++

```
struct _URB_CONTROL_FEATURE_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                Reserved2;
    ULONG                Reserved3;
    PVOID                 Reserved4;
    PMDL                 Reserved5;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    USHORT                Reserved0;
    USHORT                FeatureSelector;
    USHORT                Index;
    USHORT                Reserved1;
};
```

## Members

Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

**Hdr.Function** indicates either a set or a clear feature operation, to perform on a device, interface, endpoint or other non-standard component. **Hdr.Function** must have one of the following values:

**Hdr.Length** must equal `sizeof(_URB_CONTROL_FEATURE_REQUEST)`.

Reserved

Reserved. Do not use.

Reserved2

Reserved. Do not use.

Reserved3

Reserved. Do not use.

Reserved4

Reserved. Do not use.

Reserved5

Reserved. Do not use.

UrbLink

Reserved. Do not use.

hca

Reserved. Do not use.

Reserved0

Reserved. Do not use.

FeatureSelector

Specifies the USB-defined feature code to be cleared or set. Using a feature code that is invalid, cannot be set, or cannot be cleared will cause the target to stall. The bus driver will copy the value in the **FeatureSelector** member to the **wValue** field of the setup packet.

Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero. The bus driver will copy the value in the **Index** member to the **wIndex** field of the setup packet.

Reserved1

Reserved. Do not use.

## Remarks

Drivers can use the **UsbBuildFeatureRequest** service routine to format this URB.

The reserved members of this structure must be treated as opaque and are reserved for system use.

When a driver arms a USB device for remote wakeup with an IRP\_MN\_WAIT\_WAKE request, the USB bus driver automatically sets remote wakeup feature on the device. A control feature URB is not necessary.

Likewise, when a driver issues a URB with a function type of URB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL to a pipe, the bus driver will automatically clear the pipe's endpoint stall feature. The driver does not have to send a control feature URB to the pipe to clear the endpoint stall.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL\_GET\_CONFIGURATION\_REQUEST structure (usb.h)

Article02/22/2024

The \_URB\_CONTROL\_GET\_CONFIGURATION\_REQUEST structure is used by USB client drivers to retrieve the current configuration for a device.

## Syntax

C++

```
struct _URB_CONTROL_GET_CONFIGURATION_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                Reserved0;
    ULONG                TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                 TransferBufferMDL;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    UCHAR                Reserved1[8];
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information.

**Hdr.Function** must be set to URB\_FUNCTION\_GET\_CONFIGURATION.

**Hdr.Length** must equal `sizeof(_URB_CONTROL_GET_CONFIGURATION_REQUEST)`.

Reserved

Reserved. Do not use.

Reserved0

Reserved. Do not use.

TransferBufferLength

Must be 1. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte that specifies the index of the current configuration.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte that specifies the index of the current configuration. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### hca

Reserved. Do not use.

#### Reserved1[8]

Reserved. Do not use.

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

 Expand table

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[URB](#)

## USB Structures

### URB\_HEADER

# \_URB\_CONTROL\_GET\_INTERFACE\_REQUEST structure (usb.h)

Article02/22/2024

The \_URB\_CONTROL\_GET\_INTERFACE\_REQUEST structure is used by USB client drivers to retrieve the current alternate interface setting for an interface in the current configuration.

## Syntax

C++

```
struct _URB_CONTROL_GET_INTERFACE_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                Reserved0;
    ULONG                TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                 TransferBufferMDL;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    UCHAR                Reserved1[4];
    USHORT               Interface;
    USHORT               Reserved2;
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information. **Hdr.Function** must be [URB\\_FUNCTION\\_GET\\_INTERFACE](#), and **Hdr.Length** must equal [sizeof\(\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST\)](#).

Reserved

Reserved. Do not use.

Reserved0

Reserved. Do not use.

TransferBufferLength

Must be 1. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### hca

Reserved. Do not use.

#### Reserved1[4]

Reserved. Do not use.

#### Interface

Specifies the device-defined index of the interface descriptor being retrieved.

#### Reserved2

Reserved. Do not use.

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

 Expand table

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL\_GET\_STATUS\_REQUEST structure (usb.h)

Article02/22/2024

The \_URB\_CONTROL\_GET\_STATUS\_REQUEST structure is used by USB client drivers to retrieve status from a device, interface, endpoint, or other device-defined target.

## Syntax

C++

```
struct _URB_CONTROL_GET_STATUS_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                Reserved0;
    ULONG                TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                 TransferBufferMDL;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    UCHAR                Reserved1[4];
    USHORT               Index;
    USHORT               Reserved2;
};
```

## Members

### Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

**Hdr.Length** must be `sizeof(_URB_CONTROL_GET_STATUS_REQUEST)`, and **Hdr.Function** must be one of the following values:

- URB\_FUNCTION\_GET\_STATUS\_FROM\_DEVICE
- URB\_FUNCTION\_GET\_STATUS\_FROM\_INTERFACE
- URB\_FUNCTION\_GET\_STATUS\_FROM\_ENDPOINT
- URB\_FUNCTION\_GET\_STATUS\_FROM\_OTHER

### Reserved

Reserved. Do not use.

**Reserved0**

Reserved. Do not use.

**TransferBufferLength**

Must be 2. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

**TransferBuffer**

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte specifying the status for the target.

**TransferBufferMDL**

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte specifying the status for the target. This MDL must be allocated from nonpaged pool.

**UrbLink**

Reserved. Do not use.

**hca**

Reserved. Do not use.

**Reserved1[4]**

Reserved. Do not use.

**Index**

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero.

**Reserved2**

Reserved. Do not use.

## Remarks

Drivers can use the [UsbBuildGetStatusRequest](#) service routine to format this URB.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

 Expand table

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL\_TRANSFER structure (usb.h)

Article09/01/2022

The \_URB\_CONTROL\_TRANSFER structure is used by USB client drivers to transfer data to or from a control pipe.

## Syntax

C++

```
struct _URB_CONTROL_TRANSFER {
    struct _URB_HEADER    Hdr;
    USBD_PIPE_HANDLE      PipeHandle;
    ULONG                 TransferFlags;
    ULONG                 TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                  TransferBufferMDL;
    struct _URB            *UrbLink;
    struct _URB_HCD_AREA  hca;
    UCHAR                 SetupPacket[8];
};
```

## Members

Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

**Hdr.Function** must be URB\_FUNCTION\_CONTROL\_TRANSFER, and **Hdr.Length** must be `sizeof(_URB_CONTROL_TRANSFER)`.

PipeHandle

Handle for the control pipe.

If target is the default control endpoint, then **PipeHandle** must be NULL. In this case, the **TransferFlags** must contain the USBD\_DEFAULT\_PIPE\_TRANSFER flag.

If target is a non-default control endpoint, **PipeHandle** specifies an opaque handle for the control pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type

URB\_FUNCTION\_SELECT\_CONFIGURATION or when the client driver changes the settings for an interface with a URB of type URB\_FUNCTION\_SELECT\_INTERFACE.

#### TransferFlags

Specifies zero, one, or a combination of the following flags:

Value	Meaning
USBD_TRANSFER_DIRECTION_IN	Is set to request data from a device. To transfer data to a device, this flag must be clear.
USBD_TRANSFER_DIRECTION_OUT	Is set to transfer data to a device. Setting this flag is equivalent to clearing the USBD_TRANSFER_DIRECTION_IN flag.
USBD_SHORT_TRANSFER_OK	<p>Is set to direct the host controller not to return an error when it receives a packet from the device that is shorter than the maximum packet size for the endpoint. The maximum packet size for the endpoint is reported in the <b>bMaxPacketSize0</b> member of the <a href="#">USB_DEVICE_DESCRIPTOR</a> structure (device descriptor) for the default control endpoint. For a non-default control endpoint, maximum packet size is set in the <b>wMaxPacketSize</b> member of the <a href="#">USB_ENDPOINT_DESCRIPTOR</a> structure (endpoint descriptor).</p> <p>When the host controller receives a packet whose length is shorter than the <b>wMaxPacketSize</b> value on a control endpoint, the behavior is as follows depending on the type of host controller:</p> <ul style="list-style-type: none"><li>• On EHCI host controllers, the host controller proceeds immediately to the status phase of the control transfer. The transfer completes successfully, regardless of whether USBD_SHORT_TRANSFER_OK is set.</li><li>• On UHCI and OHCI host controllers, if USBD_SHORT_TRANSFER_OK is set, the host controller proceeds to the status phase. If USBD_SHORT_TRANSFER_OK is not set, the host controller abandons the data and status phases of the control transfer and the transfer completes with an error.</li></ul>
USBD_DEFAULT_PIPE_TRANSFER	Is set to direct the host controller to do a control transfer on the default control pipe. This allows the caller to send

commands to the default control pipe without explicitly specifying the pipe handle.

#### TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### hca

Reserved. Do not use.

#### SetupPacket[8]

Specifies a USB-defined request setup packet. The format of a USB request setup packet is found in the USB core specification.

## Remarks

The [URB\\_CONTROL\\_TRANSFER\\_EX](#) structure is identical to [URB\\_CONTROL\\_TRANSFER](#), except that it provides a timeout value in the **Timeout** field.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[URB](#)

[URB\\_CONTROL\\_TRANSFER\\_EX](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL\_TRANSFER\_EX structure (usb.h)

Article09/01/2022

The \_URB\_CONTROL\_TRANSFER\_EX structure is used by USB client drivers to transfer data to or from a control pipe, with a timeout that limits the acceptable transfer time.

## Syntax

C++

```
struct _URB_CONTROL_TRANSFER_EX {
    struct _URB_HEADER    Hdr;
    USBD_PIPE_HANDLE      PipeHandle;
    ULONG                 TransferFlags;
    ULONG                 TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                  TransferBufferMDL;
    ULONG                 Timeout;
    ULONG                 Pad;
    struct _URB_HCD_AREA hca;
    UCHAR                 SetupPacket[8];
};
```

## Members

### Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

**Hdr.Function** must be URB\_FUNCTION\_CONTROL\_TRANSFER\_EX, and **Hdr.Length** must be `sizeof(_URB_CONTROL_TRANSFER_EX)`.

### PipeHandle

Handle for the pipe.

If target is the default control endpoint, then **PipeHandle** must be NULL. In this case, the **TransferFlags** must contain the USBD\_DEFAULT\_PIPE\_TRANSFER flag.

If target is a non-default control endpoint, **PipeHandle** specifies an opaque handle for the control pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type

URB\_FUNCTION\_SELECT\_CONFIGURATION or when the client driver changes the settings for an interface with a URB of type URB\_FUNCTION\_SELECT\_INTERFACE.

#### TransferFlags

Specifies zero, one, or a combination of the following flags:

Value	Meaning
USBD_TRANSFER_DIRECTION_IN	Is set to request data from a device. To transfer data to a device, this flag must be clear.
USBD_TRANSFER_DIRECTION_OUT	Is set to transfer data to a device. Setting this flag is equivalent to clearing the USBD_TRANSFER_DIRECTION_IN flag.
USBD_SHORT_TRANSFER_OK	<p>Is set to direct the host controller not to return an error when it receives a packet from the device that is shorter than the maximum packet size for the endpoint. The maximum packet size for the endpoint is reported in the <b>bMaxPacketSize0</b> member of the <a href="#">USB_DEVICE_DESCRIPTOR</a> structure (device descriptor) for the default control endpoint. For a non-default control endpoint, maximum packet size is set in the <b>wMaxPacketSize</b> member of the <a href="#">USB_ENDPOINT_DESCRIPTOR</a> structure (endpoint descriptor).</p> <p>When the host controller receives a packet whose length is shorter than the <b>wMaxPacketSize</b> value on a control endpoint, the behavior is as follows depending on the type of host controller:</p> <ul style="list-style-type: none"><li>• On EHCI host controllers, the host controller proceeds immediately to the status phase of the control transfer. The transfer completes successfully, regardless of whether USBD_SHORT_TRANSFER_OK is set.</li><li>• On UHCI and OHCI host controllers, if USBD_SHORT_TRANSFER_OK is set, the host controller proceeds to the status phase. If USBD_SHORT_TRANSFER_OK is not set, the host controller abandons the data and status phases of the control transfer and the transfer completes with an error.</li></ul>
USBD_DEFAULT_PIPE_TRANSFER	Is set to direct the host controller to do a control transfer on the default control pipe. This allows the caller to send

commands to the default control pipe without explicitly specifying the pipe handle.

#### TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

#### Timeout

Indicates the time, in milliseconds, before the URB times out. A value of 0 indicates that there is no timeout for this URB.

#### Pad

Reserved. Do not use.

#### hca

Reserved. Do not use.

#### SetupPacket[8]

Specifies a USB-defined request setup packet. The format of a USB request setup packet is found in the USB core specification.

## Remarks

This URB structure is identical to [\\_URB\\_CONTROL\\_TRANSFER](#), except that the **Timeout** member establishes a timeout for the URB.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

<b>Minimum supported client</b>	Available in Windows Vista and later operating systems.
<b>Header</b>	usb.h (include Usb.h)

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_CONTROL\\_TRANSFER](#)

[\\_URB\\_HEADER](#)

# \_URB\_CONTROL\_VENDOR\_OR\_CLASS\_REQUEST structure (usb.h)

Article04/01/2021

The \_URB\_CONTROL\_VENDOR\_OR\_CLASS\_REQUEST structure is used by USB client drivers to issue a vendor or class-specific command to a device, interface, endpoint, or other device-defined target.

## Syntax

C++

```
struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST {
    struct _URB_HEADER    Hdr;
    PVOID                 Reserved;
    ULONG                TransferFlags;
    ULONG                TransferBufferLength;
    PVOID                 TransferBuffer;
    PMDL                 TransferBufferMDL;
    struct _URB           *UrbLink;
    struct _URB_HCD_AREA hca;
    UCHAR                RequestTypeReservedBits;
    UCHAR                Request;
    USHORT               Value;
    USHORT               Index;
    USHORT               Reserved1;
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information.

**Hdr.Function** must be one of URB\_FUNCTION\_CLASS\_XXX or URB\_FUNCTION\_VENDOR\_XXX GET\_STATUS, and **Hdr.Length** must be `sizeof(_URB_CONTROL_VENDOR_OR_CLASS_REQUEST)`.

Reserved

TransferFlags

Specifies zero, one, or a combination of the following flags:

Value	Meaning
<code>USBD_TRANSFER_DIRECTION_IN</code>	Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.
<code>USBD_SHORT_TRANSFER_OK</code>	<p>Is set to direct the host controller not to return an error when it receives a packet from the device that is shorter than the maximum packet size for the endpoint. The maximum packet size for the endpoint is reported in the <code>bMaxPacketSize0</code> member of the <a href="#">USB_DEVICE_DESCRIPTOR</a> structure (device descriptor) for the default control endpoint. For a non-default control endpoint, maximum packet size is set in the <code>wMaxPacketSize</code> member of the <a href="#">USB_ENDPOINT_DESCRIPTOR</a> structure (endpoint descriptor).</p> <p>When the host controller receives a packet whose length is shorter than the <code>wMaxPacketSize</code> value on a control endpoint, the behavior is as follows depending on the type of host controller:</p> <ul style="list-style-type: none"> <li>• On EHCI host controllers, the host controller proceeds immediately to the status phase of the control transfer. The transfer completes successfully, regardless of whether <code>USBD_SHORT_TRANSFER_OK</code> is set.</li> <li>• On UHCI and OHCI host controllers, if <code>USBD_SHORT_TRANSFER_OK</code> is set, the host controller proceeds to the status phase. If <code>USBD_SHORT_TRANSFER_OK</code> is not set, the host controller abandons the data and status phases of the control transfer and the transfer completes with an error.</li> </ul>

This flag should not be set unless `USBD_TRANSFER_DIRECTION_IN` is also set.

#### TransferBufferLength

Specifies the length, in bytes, of the buffer specified in `TransferBuffer` or described in `TransferBufferMDL`. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

#### TransferBuffer

Pointer to a resident buffer for the transfer or is `NULL` if an MDL is supplied in `TransferBufferMDL`. The contents of this buffer depend on the value of `TransferFlags`. If

**USBD\_TRANSFER\_DIRECTION\_IN** is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

#### TransferBufferMDL

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### hca

Reserved. Do not use.

#### RequestTypeReservedBits

Reserved. Do not use.

#### Request

Specifies the USB or vendor-defined request code for the device, interface, endpoint, or other device-defined target.

#### Value

Specifies a value, specific to **Request**, that becomes part of the USB-defined setup packet for the target. This value is defined by the creator of the code used in **Request**.

#### Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero.

#### Reserved1

Reserved. Do not use.

## Remarks

Drivers can use the **UsbBuildVendorRequest** service routine format this URB.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_GET\_CURRENT\_FRAME\_NUMBER structure (usb.h)

Article02/22/2024

The \_URB\_GET\_CURRENT\_FRAME\_NUMBER structure is used by USB client drivers to retrieve the current frame number.

## Syntax

C++

```
struct _URB_GET_CURRENT_FRAME_NUMBER {
    struct _URB_HEADER Hdr;
    ULONG             FrameNumber;
};
```

## Members

Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information.

Hdr.Function must be URB\_FUNCTION\_GET\_CURRENT\_FRAME\_NUMBER, and

Hdr.Length must be `sizeof(_URB_GET_CURRENT_FRAME_NUMBER)`.

FrameNumber

Contains the current 32-bit frame number, on the USB bus, on return from the host controller driver.

## Requirements

  Expand table

Requirement	Value
Header	usb.h (include Usb.h)

## See also

URB

USB Structures

\_URB\_HEADER

# \_URB\_GET\_ISOCH\_PIPE\_TRANSFER\_PATH\_DELAYS structure (usb.h)

Article02/22/2024

The \_URB\_GET\_ISOCH\_PIPE\_TRANSFER\_PATH\_DELAYS structure is used by USB client drivers to retrieve delays associated with isochronous transfer programming in the host controller and transfer completion so that the client driver can ensure that the device gets the isochronous packets in time.

## Syntax

C++

```
struct _URB_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS {
    struct _URB_HEADER Hdr;
    USBD_PIPE_HANDLE PipeHandle;
    ULONG MaximumSendPathDelayInMilliSeconds;
    ULONG MaximumCompletionPathDelayInMilliSeconds;
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information. **Hdr.Function** must be `URB_FUNCTION_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS`, and **Hdr.Length** must be set to `sizeof(_URB_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS)`.

PipeHandle

Specifies an opaque handle to pipe associated with the endpoint. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type `URB_FUNCTION_SELECT_CONFIGURATION` or when the client driver changes the settings for an interface with a URB of type `URB_FUNCTION_SELECT_INTERFACE`.

MaximumSendPathDelayInMilliSeconds

Returns the maximum delay in milliseconds from the time the client driver's isochronous transfer is received by the USB driver stack to the time the transfer is programmed in the host controller. The host controller could either be a local host (as in case of wired USB)

or it could be a remote controller as in case of Media-Agnostic USB (MA-USB). In case of MA-USB, it includes the maximum delay associated with the network medium.

#### MaximumCompletionPathDelayInMilliSeconds

Returns the maximum delay in milliseconds from the time an isochronous transfer is completed by the (local or remote) host controller to the time the corresponding client driver's request is completed by the USB driver stack. For MA-USB, it includes the maximum delay associated with the network medium.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usb.h

# **\_URB\_HEADER structure (usb.h)**

Article 04/01/2021

The **\_URB\_HEADER** structure is used by USB client drivers to provide basic information about the request being sent to the host controller driver.

## Syntax

C++

```
struct _URB_HEADER {
    USHORT Length;
    USHORT Function;
    USBD_STATUS Status;
    PVOID UsbdDeviceHandle;
    ULONG UsbdFlags;
};
```

## Members

**Length**

Specifies the length, in bytes, of the URB. For URB requests that use data structures other than **\_URB\_HEADER**, this member must be set to the length of the entire URB request structure, not the **\_URB\_HEADER** size.

**Function**

Specifies a numeric code indicating the requested operation for this URB. One of the following values must be set:

### **URB\_FUNCTION\_SELECT\_CONFIGURATION**

Indicates to the host controller driver that a configuration is to be selected. If set, the URB is used with [URB\\_SELECT\\_CONFIGURATION](#) as the data structure.

### **URB\_FUNCTION\_SELECT\_INTERFACE**

Indicates to the host controller driver that an alternate interface setting is being selected for an interface. If set, the URB is used with [\\_URB\\_SELECT\\_INTERFACE](#) as the data structure.

## **URB\_FUNCTION\_ABORT\_PIPE**

Indicates that all outstanding requests for a pipe should be canceled. If set, the URB is used with [\\_URB\\_PIPE\\_REQUEST](#) as the data structure. This general-purpose request enables a client to cancel any pending transfers for the specified pipe. Pipe state and endpoint state are unaffected. The abort request might complete before all outstanding requests have completed. Do *not* assume that completion of the abort request implies that all other outstanding requests have completed.

## **URB\_FUNCTION\_TAKE\_FRAME\_LENGTH\_CONTROL**

This URB function is **deprecated** in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use. If you specify this function with an URB request, the request will fail and the system will report an error.

## **URB\_FUNCTION\_RELEASE\_FRAME\_LENGTH\_CONTROL**

This URB function is **deprecated** in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use. If you specify this function with an URB request, the request will fail and the system will report an error.

## **URB\_FUNCTION\_GET\_FRAME\_LENGTH**

This URB function is **deprecated** in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use. If you use this function with a URB request, the request will fail and the system will report an error.

## **URB\_FUNCTION\_SET\_FRAME\_LENGTH**

This URB function is **deprecated** in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use. If you use it with a URB request, the request will fail and the system will report an error.

## **URB\_FUNCTION\_GET\_CURRENT\_FRAME\_NUMBER**

Requests the current frame number from the host controller driver. If set, the URB is used with [\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#) as the data structure.

## **URB\_FUNCTION\_CONTROL\_TRANSFER**

Transfers data to or from a control pipe. If set, the URB is used with [\\_URB\\_CONTROL\\_TRANSFER](#) as the data structure.

## URB\_FUNCTION\_CONTROL\_TRANSFER\_EX

Transfers data to or from a control pipe without a time limit specified by a timeout value. If set, the URB is used with [URB\\_CONTROL\\_TRANSFER\\_EX](#) as the data structure.

Available in Windows Vista and later operating systems.

## URB\_FUNCTION\_BULK\_OR\_INTERRUPT\_TRANSFER

Transfers data from a bulk pipe or interrupt pipe or to an bulk pipe. If set, the URB is used with [\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#) as the data structure.

## URB\_FUNCTION\_BULK\_OR\_INTERRUPT\_TRANSFER\_USING\_CHAINED\_MDL

Transfers data to and from a bulk pipe or interrupt pipe, by using chained MDLs. If set, the URB is used with [\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#) as the data structure. The client driver must set the **TransferBufferMDL** member to the first [MDL](#) structure in the chain that contains the transfer buffer. The USB driver stack ignores the **TransferBuffer** member when processing this URB.

Available in Windows 8. For information about using chained MDLs see [How to Send Chained MDLs](#)">How to Send Chained MDLs.

## URB\_FUNCTION\_ISOCH\_TRANSFER

Transfers data to or from an isochronous pipe. If set, the URB is used with [\\_URB\\_ISOCH\\_TRANSFER](#) as the data structure.

## URB\_FUNCTION\_ISOCH\_TRANSFER\_USING\_CHAINED\_MDL

Transfers data to or from an isochronous pipe by using chained MDLs. If set, the URB is used with [\\_URB\\_ISOCH\\_TRANSFER](#) as the data structure. The client driver must set the **TransferBufferMDL** member to the first [MDL](#) in the chain that contains the transfer buffer. The USB driver stack ignores the **TransferBuffer** member when processing this URB.

Available in Windows 8. For information about using chained MDLs see [How to Send Chained MDLs](#)">How to Send Chained MDLs.

## URB\_FUNCTION\_RESET\_PIPE

See URB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL.

## URB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL

Resets the indicated pipe. If set, this URB is used with [\\_URB\\_PIPE\\_REQUEST](#).

ⓘ Note

This URB replaces URB\_FUNCTION\_RESET\_PIPE.

The bus driver accomplishes three tasks in response to this URB:

First, for all pipes except isochronous pipes, this URB sends a CLEAR\_FEATURE request to clear the device's ENDPOINT\_HALFT feature.

Second, the USB bus driver resets the data toggle on the host side, as required by the USB specification. The USB device should reset the data toggle on the device side when the bus driver clears its ENDPOINT\_HALFT feature. Since some non-compliant devices do not support this feature, Microsoft provides the two additional URBs:

URB\_FUNCTION\_SYNC\_CLEAR\_STALL and URB\_FUNCTION\_SYNC\_RESET\_PIPE. These allow client drivers to clear the ENDPOINT\_HALFT feature on the device, or reset the pipe on the host side, respectively, without affecting the data toggle on the host side. If the device does not reset the data toggle when it should, then the client driver can compensate for this defect by not resetting the host-side data toggle. If the data toggle is reset on the host side but not on the device side, packets will get out of sequence, and the device might drop packets.

Third, after the bus driver has successfully reset the pipe, it resumes transfers with the next queued URB.

After a pipe reset, transfers resume with the next queued URB.

It is not necessary to clear a halt condition on a default control pipe. The default control pipe must always accept setup packets, and so if it halts, the USB stack will clear the halt condition automatically. The client driver does not need to take any special action to clear the halt condition on a default pipe.

All transfers must be aborted or canceled before attempting to reset the pipe.

This URB must be sent at PASSIVE\_LEVEL.

## URB\_FUNCTION\_SYNC\_RESET\_PIPE

Clears the halt condition on the host side of a pipe. If set, this URB is used with [\\_URB\\_PIPE\\_REQUEST](#) as the data structure.

This URB allows a client to clear the halted state of a pipe without resetting the data toggle and without clearing the endpoint stall condition (feature ENDPOINT\_HALT). To clear a halt condition on the pipe, reset the host-side data toggle and clear a stall on the device with a single operation, use

URB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL.

The following status codes are important and have the indicated meaning:

USBD\_STATUS\_INVALID\_PIPE\_HANDLE

The **PipeHandle** is not valid

USBD\_STATUS\_ERROR\_BUSY

The endpoint has active transfers pending.

It is not necessary to clear a halt condition on a default control pipe. The default control pipe must always accept setup packets, and so if it halts, the USB stack will clear the halt condition automatically. The client driver does not need to take any special action to clear the halt condition on a default pipe.

All transfers must be aborted or canceled before attempting to reset the pipe.

This URB must be sent at PASSIVE\_LEVEL.

Available in Windows XP and later operating systems.

## URB\_FUNCTION\_SYNC\_CLEAR\_STALL

Clears the stall condition on the endpoint. For all pipes except isochronous pipes, this URB sends a CLEAR\_FEATURE request to clear the device's ENDPOINT\_HALT feature. However, unlike the RB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL function, this URB function does not reset the data toggle on the host side of the pipe. The USB specification requires devices to reset the device-side data toggle after the client clears the device's ENDPOINT\_HALT feature, but some non-compliant devices do not reset their data toggle properly. Client drivers that manage such devices can compensate for this defect by clearing the stall condition directly with URB\_FUNCTION\_SYNC\_CLEAR\_STALL instead of resetting the pipe with URB\_FUNCTION\_SYNC\_RESET\_PIPE\_AND\_CLEAR\_STALL. URB\_FUNCTION\_SYNC\_CLEAR\_STALL clears a stall condition on the device without

resetting the host-side data toggle. This prevents a non-compliant device from interpreting the next packet as a retransmission and dropping the packet.

If set, the URB is used with [\\_URB\\_PIPE\\_REQUEST](#) as the data structure.

This URB function should be sent at PASSIVE\_LEVEL

Available in Windows XP and later operating systems.

## **URB\_FUNCTION\_GET\_DESCRIPTOR\_FROM\_DEVICE**

Retrieves the device descriptor from a specific USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET\_DESCRIPTOR\_FROM\_ENDPOINT**

Retrieves the descriptor from an endpoint on an interface for a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_DESCRIPTOR\_TO\_DEVICE**

Sets a device descriptor on a device. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_DESCRIPTOR\_TO\_ENDPOINT**

Sets an endpoint descriptor on an endpoint for an interface. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_FEATURE\_TO\_DEVICE**

Sets a USB-defined feature on a device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_FEATURE\_TO\_INTERFACE**

Sets a USB-defined feature on an interface for a device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_FEATURE\_TO\_ENDPOINT**

Sets a USB-defined feature on an endpoint for an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET FEATURE\_TO\_OTHER**

Sets a USB-defined feature on a device-defined target on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLEAR FEATURE\_TO\_DEVICE**

Clears a USB-defined feature on a device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLEAR FEATURE\_TO\_INTERFACE**

Clears a USB-defined feature on an interface for a device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLEAR FEATURE\_TO\_ENDPOINT**

Clears a USB-defined feature on an endpoint, for an interface, on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLEAR FEATURE\_TO\_OTHER**

Clears a USB-defined feature on a device defined target on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET STATUS\_FROM DEVICE**

Retrieves status from a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET STATUS\_FROM\_INTERFACE**

Retrieves status from an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET STATUS\_FROM\_ENDPOINT**

Retrieves status from an endpoint for an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET\_STATUS\_FROM\_OTHER**

Retrieves status from a device-defined target on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_VENDOR\_DEVICE**

Sends a vendor-specific command to a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_VENDOR\_INTERFACE**

Sends a vendor-specific command for an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_VENDOR\_ENDPOINT**

Sends a vendor-specific command for an endpoint on an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_VENDOR\_OTHER**

Sends a vendor-specific command to a device-defined target on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLASS\_DEVICE**

Sends a USB-defined class-specific command to a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLASS\_INTERFACE**

Sends a USB-defined class-specific command to an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLASS\_ENDPOINT**

Sends a USB-defined class-specific command to an endpoint, on an interface, on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_CLASS\_OTHER**

Sends a USB-defined class-specific command to a device defined target on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET\_CONFIGURATION**

Retrieves the current configuration on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_CONFIGURATION\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET\_INTERFACE**

Retrieves the current settings for an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST](#) as the data structure.

Available in Windows 2000, and Windows Vista and later operating systems. Not available in Windows XP.

## **URB\_FUNCTION\_GET\_DESCRIPTOR\_FROM\_INTERFACE**

Retrieves the descriptor from an interface for a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_SET\_DESCRIPTOR\_TO\_INTERFACE**

Sets a descriptor for an interface on a USB device. If set, the URB is used with [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

## **URB\_FUNCTION\_GET\_MS\_FEATURE\_DESCRIPTOR**

Retrieves a Microsoft OS feature descriptor from a USB device or an interface on a USB device. If set, the URB is used with [\\_URB\\_OS\\_FEATURE\\_DESCRIPTOR\\_REQUEST](#) as the data structure.

Available in Windows XP and later operation systems.

## **URB\_FUNCTION\_OPEN\_STATIC\_STREAMS**

Opens streams in the specified bulk endpoint. If set, the URB is used with [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#) as the data structure.

Available in Windows 8. For information about formatting an URB for an open-stream request, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

## **URB\_FUNCTION\_CLOSE\_STATIC\_STREAMS**

Closes all opened streams in the specified bulk endpoint. If set, the URB is used with [\\_URB\\_PIPE\\_REQUEST](#) as the data structure.

Available in Windows 8. For information about formatting an URB for an open-stream request, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

### **Status**

Contains a USBD\_STATUS\_XXX code on return from the host controller driver.

### **UsbdDeviceHandle**

Reserved. Do not use.

### **UsbdFlags**

Reserved. Do not use.

## **Remarks**

The [\\_URB\\_HEADER](#) structure is a member of all USB requests that are part of the URB structure. The [\\_URB\\_HEADER](#) structure is used to provide common information about each request to the host controller driver.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## **Requirements**

Header	usb.h (include Usb.h)

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

[\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#)

[\\_URB\\_CONTROL\\_GET\\_CONFIGURATION\\_REQUEST](#)

[\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST](#)

[\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#)

[\\_URB\\_CONTROL\\_TRANSFER](#)

[\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#)

[\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#)

[\\_URB\\_ISOCH\\_TRANSFER](#)

[\\_URB\\_OS\\_FEATURE\\_DESCRIPTOR\\_REQUEST](#)

[\\_URB\\_PIPE\\_REQUEST](#)

[\\_URB\\_SELECT\\_CONFIGURATION](#)

[\\_URB\\_SELECT\\_INTERFACE](#)

# \_URB\_ISOCH\_TRANSFER structure (usb.h)

Article09/01/2022

The \_URB\_ISOCH\_TRANSFER structure is used by USB client drivers to send data to or retrieve data from an isochronous transfer pipe.

## Syntax

C++

```
struct _URB_ISOCH_TRANSFER {
    struct _URB_HEADER          Hdr;
    USBD_PIPE_HANDLE            PipeHandle;
    ULONG                        TransferFlags;
    ULONG                        TransferBufferLength;
    PVOID                        TransferBuffer;
    PMDL                         TransferBufferMDL;
    struct _URB                  *UrbLink;
    struct _URB_HCD_AREA         hca;
    ULONG                        StartFrame;
    ULONG                        NumberOfPackets;
    ULONG                        ErrorCount;
    USBD_ISO_PACKET_DESCRIPTOR  IsoPacket[1];
};
```

## Members

Hdr

A pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information.

**Hdr.Function** must be URB\_FUNCTION\_ISOCH\_TRANSFER, and **Hdr.Length** must be the size of this variable-length data structure.

PipeHandle

Specifies an opaque handle to the isochronous pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type URB\_FUNCTION\_SELECT\_CONFIGURATION or when the client driver changes the settings for an interface with a URB of type URB\_FUNCTION\_SELECT\_INTERFACE.

TransferFlags

Specifies zero, one, or a combination of the following flags:

Value	Meaning
<code>USBD_TRANSFER_DIRECTION_IN</code>	Is set to request data from a device. To transfer data to a device, this flag must be clear.
<code>USBD_SHORT_TRANSFER_OK</code>	Is set to direct the host controller not to return an error when it receives a packet from the device that is shorter than the maximum packet size for the endpoint. This flag has no effect on an isochronous pipe, because the bus driver does not return an error when it receives short packets on an isochronous pipe.
<code>USBD_START_ISO_TRANSFER_ASAP</code>	Causes the transfer to begin on the next frame, if no transfers have been submitted to the pipe since the pipe was opened or last reset. Otherwise, the transfer begins on the first frame that follows all currently queued requests for the pipe. The actual frame that the transfer begins on will be adjusted for bus latency by the host controller driver.

#### TransferBufferLength

Specifies the length, in bytes, of the buffer specified in `TransferBuffer` or described in `TransferBufferMDL`. The host controller driver returns the number of bytes that are sent to or read from the pipe in this member.

#### TransferBuffer

A pointer to a resident buffer for the transfer is **NULL** if an MDL is supplied in `TransferBufferMDL`. The contents of this buffer depend on the value of `TransferFlags`. If `USBD_TRANSFER_DIRECTION_IN` is specified, this buffer will contain data that is read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

#### TransferBufferMDL

A pointer to an MDL that describes a resident buffer is **NULL** if a buffer is supplied in `TransferBuffer`. The contents of the buffer depend on the value of `TransferFlags`. If `USBD_TRANSFER_DIRECTION_IN` is specified, the described buffer will contain data that is read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

#### UrbLink

Reserved. Do not use.

#### **hca**

Reserved. Do not use.

#### **StartFrame**

Specifies the frame number that the transfer should begin on. This variable must be within a system-defined range of the current frame. The range is specified by the constant [USBD\\_ISO\\_START\\_FRAME\\_RANGE](#).

If [START\\_ISO\\_TRANSFER\\_ASAP](#) is set in [TransferFlags](#), this member contains the frame number that the transfer began on, when the request is returned by the host controller driver. Otherwise, this member must contain the frame number that this transfer begins on.

#### **NumberOfPackets**

Specifies the number of packets that are described by the variable-length array member [IsoPacket](#).

#### **ErrorCount**

Contains the number of packets that completed with an error condition on return from the host controller driver.

#### **IsoPacket[1]**

Contains a variable-length array of [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#) structures that describe the isochronous transfer packets to be transferred on the USB bus. For more information about this member see the Remarks section.

## **Remarks**

The USB bus driver always returns a value of [USBD\\_STATUS\\_SUCCESS](#) in [Hdr.Status](#), unless every packet in the transfer generated an error or the request was not well-formed and could not be executed at all. The following table includes possible error codes returned in [Hdr.Status](#).

<b>Error value</b>	<b>Meaning</b>
<a href="#">USBD_STATUS_ISOCH_REQUEST_FAILED</a>	Indicates that every packet of an isochronous request was completed with errors.

USBD_STATUS_BAD_START_FRAME	Indicates that the requested start frame is not within USBD_ISO_START_FRAME_RANGE of the current USB frame.
USBD_ISO_NOT_ACCESSED_LATE	Indicates that every packet was submitted too late for the packet to be sent, based on the requested start frame.
USBD_STATUS_INVALID_PARAMETER	Indicates that one of the URB parameters was incorrect.

Before the host controller sends an isochronous request to a USB device, it requires information about the device's endpoint to which it must send or receive data. This information is stored in endpoint descriptors ([USB\\_ENDPOINT\\_DESCRIPTOR](#)) that are retrieved from the selected configuration descriptor. After the bus driver gets the endpoint descriptor, it creates an isochronous transfer pipe to set up the data transfer. The pipe's attributes are stored in the [USBD\\_PIPE\\_INFORMATION](#) structure. For isochronous transfers, the members are set as follows:

- The **PipeType** member specifies the type of transfer and is set to UsbdPipeTypeIsochronous.
- The **MaximumPacketSize** member specifies the amount of data, in bytes, that constitutes one packet. For isochronous transfers, the packet size is fixed and can be a value from 0-1024. The packet size either equals or is less than the **wMaxPacketSize** value of the endpoint descriptor.
- The **Interval** member is derived from the **bInterval** value of the endpoint descriptor. This value is used to calculate the polling period that indicates the frequency at which data is sent on the bus. For full speed devices, the period is measured in units of 1 millisecond frames; for high speed devices, the period is measured in microframes.

The host controller also determines the amount of data that can be transferred (within a frame or a microframe) depending on the type of device. This information is available in bits 12.. 11 of **wMaxPacketSize** in the endpoint descriptor.

For full-speed devices, only one packet can be transferred within a frame; bits 12.. 11 are reserved and set to zero.

For high speed devices, data can be transferred in a single packet or might span multiple packets, within a microframe. If bits 12.. 11 are set to  $n$ , you can transfer  $(n+1)*\text{MaximumPacketSize}$  bytes per microframe. Bits 12.. 11 set to zero indicate that only one packet can be transferred in a microframe. If bits 12.. 11 are set to 1, the host controller can transfer two packets in a microframe.

The **IsoPacket** member of `_URB_ISOCH_TRANSFER` is an array of `USBD_ISO_PACKET_DESCRIPTOR` that describes the transfer buffer layout. Each element in the array correlates to data that is transferred in one microframe. If **IsoPacket** has  $n$  elements, the host controller transfers use  $n$  frames (for full speed devices) or microframes (for high speed devices) to transfer data. The `IsoPacket[i].Offset` member is used to track the amount of data to send or receive. This is done by setting a byte offset from the start of the entire transfer buffer for the request.

For example, there are five microframes available to transfer 1024 byte-sized packets.

If bits 12.. 11 are set to zero (indicating a single packet per microframe transfer), **IsoPacket** contains the following entries:

Microframe 1 `IsoPacket.Element[0].Offset = 0` (start address)

Microframe 2 `IsoPacket.Element[1].Offset = 1024`

Microframe 3 `IsoPacket.Element[2].Offset = 2048`

Microframe 4 `IsoPacket.Element[3].Offset = 3072`

Microframe 5 `IsoPacket.Element[4].Offset = 4096`

If bits 12.. 11 are set to 1 (indicating two packets per microframe), **IsoPacket** contains the following entries:

Microframe 1 `IsoPacket.Element[0].Offset = 0` (start address)

Microframe 2 `IsoPacket.Element[1].Offset = 2048`

Microframe 3 `IsoPacket.Element[2].Offset = 4096`

Microframe 4 `IsoPacket.Element[3].Offset = 6144`

Microframe 5 `IsoPacket.Element[4].Offset = 8192`

**Note** For multiple packets, the offset value indicates sizes for all the packets within the microframe.

The `IsoPacket[i].Length` member is updated by the host controller to indicate the actual number of bytes that are received from the device for isochronous IN transfers.

`IsoPacket[i].Length` is not used for isochronous OUT transfers.

Drivers can use the `GET_ISO_URB_SIZE` macro to determine the size that is needed to hold the entire URB. If the length is too small to fill the space set aside for this packet,

the bus driver leaves a gap from the end of the retrieved data to the offset for the next packet. The bus driver will not adjust the offsets to avoid wasting buffer space.

The **TransferBuffer** or **TransferBufferMDL** members must specify a virtually contiguous buffer.

Treat other members that are part of this structure but not described here as opaque. They are reserved for system use.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[How to Transfer Data to USB Isochronous Endpoints](#)

[URB](#)

[USB Structures](#)

[USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#)

[USBD\\_IsochUrbAllocate](#)

[\\_URB\\_HEADER](#)

# \_URB\_OPEN\_STATIC\_STREAMS structure (usb.h)

Article04/01/2021

The \_URB\_OPEN\_STATIC\_STREAMS structure is used by a USB client driver to open streams in the specified bulk endpoint.

To format the URB, call the [UsbBuildOpenStaticStreamsRequest](#) function.

## Syntax

C++

```
struct _URB_OPEN_STATIC_STREAMS {
    struct _URB_HEADER          Hdr;
    USBD_PIPE_HANDLE            PipeHandle;
    ULONG                       NumberOfStreams;
    USHORT                      StreamInfoVersion;
    USHORT                      StreamInfoSize;
    PUSBD_STREAM_INFORMATION   Streams;
};
```

## Members

Hdr

The [\\_URB\\_HEADER](#) structure that specifies the URB header information. **Hdr.Function** must be [URB\\_FUNCTION\\_OPEN\\_STATIC\\_STREAMS](#), and **Hdr.Length** must be [sizeof\(\\_URB\\_OPEN\\_STATIC\\_STREAMS\)](#).

PipeHandle

An opaque handle for the pipe associated with the endpoint that supports the streams to open.

The client driver obtains **PipeHandle** from the [URB\\_FUNCTION\\_SELECT\\_CONFIGURATION](#) or [URB\\_FUNCTION\\_SELECT\\_INTERFACE](#) request.

NumberOfStreams

The number of streams to open. The **NumberOfStreams** value indicates the number of elements in the array pointed to by **Streams**. This value must be greater than zero and less than or equal to the maximum number of streams supported by the USB driver stack, the host controller, and the endpoint in the device. For more information, see Remarks.

#### StreamInfoVersion

Version of the [USBD\\_STREAM\\_INFORMATION](#) structure. Must be set to URB\_OPEN\_STATIC\_STREAMS\_VERSION\_100; otherwise, the request fails and the URB status is USBD\_STATUS\_INVALID\_PARAMETER.

#### StreamInfoSize

Size of the [USBD\\_STREAM\\_INFORMATION](#) structure. **StreamInfoSize** must be `sizeof(USBD_STREAM_INFORMATION)`; otherwise, the request fails and the URB status is USBD\_STATUS\_INFO\_LENGTH\_MISMATCH.

#### Streams

Pointer to a caller-allocated, initialized array of [USBD\\_STREAM\\_INFORMATION](#) structures. The length of the array depends on the number of streams to open and must be the same as the **NumberOfStreams** value. For more information, see Remarks.

## Remarks

To use streams (other than the default stream) in the endpoint for I/O operations, the client driver opens the required streams by sending an open-stream request (URB\_FUNCTION\_OPEN\_STATIC\_STREAMS) to the USB driver stack. For the request, the client driver must format the URB by initializing the [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#) structure. To format the URB, call the [UsbBuildOpenStaticStreamsRequest](#) function.

The maximum number of streams that can be opened by a client driver, must be less than or equal to the maximum number of streams supported by the USB driver stack, the host controller, and the bulk endpoint. To get the maximum number of streams supported by the host controller, call [USBD\\_QueryUsbCapability](#). The USB driver stack supports up to 255 streams. If the client driver requests more than 255 streams, [USBD\\_QueryUsbCapability](#) fails the request. To get the maximum number of streams supported by the endpoint, inspect the endpoint companion descriptor (see [USB\\_SUPERSPEED\\_ENDPOINT\\_COMPANION\\_DESCRIPTOR](#) in Usbspec.h).

For information about formatting the URB for the open-stream request and code example, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

When the client driver is finished using the streams, the driver can close all streams associated with a particular endpoint by sending a close-stream request. To send the request, the client driver must specify information about the endpoint in the [\\_URB\\_PIPE\\_REQUEST](#) structure. 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 streams in use.

## Requirements

Minimum supported client	Windows 8
Header	usb.h

## See also

[How to Open and Close Static Streams in a USB Bulk Endpoint](#)

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_OS\_FEATURE\_DESCRIPTOR\_REQUEST structure (usb.h)

Article04/01/2021

The \_URB\_OS\_FEATURE\_DESCRIPTOR\_REQUEST structure is used by the USB hub driver to retrieve Microsoft OS Feature Descriptors from a USB device or an interface on a USB device.

## Syntax

C++

```
struct _URB_OS FEATURE_DESCRIPTOR REQUEST {
    struct _URB HEADER Hdr;
    PVOID Reserved;
    ULONG Reserved0;
    ULONG TransferBufferLength;
    PVOID TransferBuffer;
    PMDL TransferBufferMDL;
    struct _URB *UrbLink;
    struct _URB_HCD AREA hca;
    UCHAR Recipient : 5;
    UCHAR Reserved1 : 3;
    UCHAR Reserved2;
    UCHAR InterfaceNumber;
    UCHAR MS_PageIndex;
    USHORT MS_FeatureDescriptorIndex;
    USHORT Reserved3;
};
```

## Members

Hdr

Pointer to a \_URB\_HEADER structure that specifies the URB header information. **Hdr.Function** must be URB\_FUNCTION\_GET\_MS\_FEATURE\_DESCRIPTOR. **Hdr.Length** must be `sizeof(_URB_OS_FEATURE_DESCRIPTOR_REQUEST)`.

Reserved

Reserved0

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes read in this member. Current implementation of this function limits the maximum MS OS Feature Descriptor size to 4 Kilobytes.

#### **TransferBuffer**

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in **TransferBufferMDL**.

#### **TransferBufferMDL**

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in **TransferBuffer**. This MDL must be allocated from nonpaged pool.

#### **UrbLink**

Reserved. Do not use.

#### **hca**

#### **Recipient**

Specifies whether the recipient is the USB device or an interface on the USB device. One of the following values must be specified:

- 0 indicates that the USB device is the recipient of the request.
- 1 indicates that a USB interface is the recipient of the request.
- 2 indicates that a USB endpoint is the recipient of the request.

#### **Reserved1**

#### **Reserved2**

#### **InterfaceNumber**

Indicates the interface number that is the recipient of the request, if the **Recipient** member value is 1. Must be set to 0 if the USB device is the recipient.

#### **MS\_PageIndex**

Must be set to 0. Page index of the 64K page of the MS OS Feature Descriptor to be returned. Current implementation only supports a maximum descriptor size of 4K.

#### **MS\_FeatureDescriptorIndex**

Index for MS OS Feature Descriptor to be requested.

Reserved3

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

<b>Minimum supported client</b>	Available in Windows XP and later operating systems.
<b>Header</b>	usb.h (include Usb.h)

## See also

[URB](#)

[\\_URB\\_HEADER](#)

# \_URB\_PIPE\_REQUEST structure (usb.h)

Article02/22/2024

The `_URB_PIPE_REQUEST` structure is used by USB client drivers to clear a stall condition on an endpoint.

## Syntax

C++

```
struct _URB_PIPE_REQUEST {
    struct _URB_HEADER Hdr;
    USBD_PIPE_HANDLE PipeHandle;
    ULONG             Reserved;
};
```

## Members

Hdr

Pointer to the `_URB_HEADER` structure that specifies the URB header information.

`Hdr.Function` must be one of the following:

- `URB_FUNCTION_SYNC_RESET_PIPE_AND_CLEAR_STALL`
- `URB_FUNCTION_SYNC_RESET_PIPE`
- `URB_FUNCTION_SYNC_CLEAR_STALL`
- `URB_FUNCTION_ABORT_PIPE`
- `URB_FUNCTION_CLOSE_STATIC_STREAMS`

The `Hdr.Length` member must be `sizeof(_URB_PIPE_REQUEST)`.

PipeHandle

Specifies an opaque handle to the bulk or interrupt pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type `URB_FUNCTION_SELECT_CONFIGURATION` or when the client driver changes the settings for an interface with a URB of type `URB_FUNCTION_SELECT_INTERFACE`.

Reserved

Reserved. Do not use.

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

 Expand table

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

# \_URB\_SELECT\_CONFIGURATION structure (usb.h)

Article04/01/2021

The \_URB\_SELECT\_CONFIGURATION structure is used by client drivers to select a configuration for a USB device.

## Syntax

C++

```
struct _URB_SELECT_CONFIGURATION {
    struct _URB_HEADER             Hdr;
    PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor;
    USBD_CONFIGURATION_HANDLE      ConfigurationHandle;
    USBD_INTERFACE_INFORMATION     Interface;
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information. **Hdr.Function** must be **URB\_FUNCTION\_SELECT\_CONFIGURATION**, and **Hdr.Length** must be the size of the entire URB. Drivers may use the **GET\_SELECT\_CONFIGURATION\_REQUEST\_SIZE** macro defined in **usbdlib.h** to obtain the size of the URB.

ConfigurationDescriptor

Pointer to an initialized USB configuration descriptor that identifies the configuration to be used on the device. If this member is **NULL**, the device will be set to an unconfigured state.

ConfigurationHandle

Contains a handle that is used to access this configuration on return from the host controller driver. USB client drivers must treat this member as opaque.

Interface

Specifies a variable length array of [USBD\\_INTERFACE\\_INFORMATION](#) structures, each describing an interface supported by the configuration being selected.

Before the request is sent to the host controller driver, the driver may select an alternate setting for one or more of the interfaces contained in this array by setting members of the [USBD\\_INTERFACE\\_INFORMATION](#) structure for that interface.

On return from the host controller driver, this member contains a [USBD\\_INTERFACE\\_INFORMATION](#) structure with data describing the capabilities and format of the endpoints within that interface.

## Remarks

An URB\_FUNCTION\_SELECT\_CONFIGURATION URB consists of a [\\_URB\\_SELECT\\_CONFIGURATION](#) structure followed by a sequence of variable-length array of [USBD\\_INTERFACE\\_INFORMATION](#) structures, each element in the array for each unique interface number in the configuration. Client drivers must allocate enough memory to contain one [USBD\\_PIPE\\_INFORMATION](#) structure for each endpoint in the selected interfaces.

Drivers can use the [USBD\\_CreateConfigurationRequestEx](#) service routine to allocate the URB.

Other members that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[URB](#)

[USB Structures](#)

[USBD\\_CreateConfigurationRequestEx](#)

[USBD\\_INTERFACE\\_INFORMATION](#)

USBD\_PIPE\_INFORMATION

\_URB\_HEADER

# \_URB\_SELECT\_INTERFACE structure (usb.h)

Article04/01/2021

The \_URB\_SELECT\_INTERFACE structure is used by USB client drivers to select an alternate setting for an interface or to change the maximum packet size of a pipe in the current configuration on a USB device.

## Syntax

C++

```
struct _URB_SELECT_INTERFACE {
    struct _URB_HEADER      Hdr;
    USBD_CONFIGURATION_HANDLE ConfigurationHandle;
    USBD_INTERFACE_INFORMATION Interface;
};
```

## Members

Hdr

Pointer to a [\\_URB\\_HEADER](#) structure that specifies the URB header information. **Hdr.Function** must be [URB\\_FUNCTION\\_SELECT\\_INTERFACE](#), and **Hdr.Length** must be the size of the entire URB.

ConfigurationHandle

Specifies the handle to the configuration that this interface belongs to. The host controller driver returns this handle when the client selects the configuration with an [URB\\_FUNCTION\\_SELECT\\_CONFIGURATION](#) request.

Interface

A variable-length [USBD\\_INTERFACE\\_INFORMATION](#) structure that specifies the interface and the new alternate setting for that interface, and if required, the new maximum packet sizes for the corresponding pipes. For more information, see Remarks.

## Remarks

You can use the [GET\\_SELECT\\_INTERFACE\\_REQUEST\\_SIZE](#) macro to determine the size of the URB\_FUNCTION\_SELECT\_INTERFACE URB, and the [UsbBuildSelectInterfaceRequest](#) routine to format the URB.

The [USBD\\_INTERFACE\\_INFORMATION](#) structure contains information about the interface and its alternate setting. The [Pipes](#) member of [USBD\\_INTERFACE\\_INFORMATION](#) points to an array of [USBD\\_PIPE\\_INFORMATION](#) structures. The array stores information about the pipes associated with the endpoints of the interface. You can override certain default settings for a pipe, such as its maximum packet size. To alter the maximum packet size, set the [USBD\\_PFF\\_CHANGE\\_MAX\\_PACKET](#) flag in [Pipes\[i\].PipeFlags](#), and then specify the new value in [Pipes\[i\].MaximumPacketSize](#).

After the bus driver successfully completes processing the URB\_FUNCTION\_SELECT\_INTERFACE URB, it returns an array of handles for each pipe in the [Pipes\[i\].PipeHandle](#) member. The client driver can store pipe handles to send I/O requests to specific pipes.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[Configuring USB Devices](#)

[URB](#)

[USB Structures](#)

[USBD\\_INTERFACE\\_INFORMATION](#)

[\\_URB\\_HEADER](#)

# URB structure (usb.h)

Article04/01/2021

The **URB** structure is used by USB client drivers to describe USB request blocks (URBs) that send requests to the USB driver stack. The **URB** structure defines a format for all possible commands that can be sent to a USB device.

## Syntax

C++

```
typedef struct _URB {
    union {
#ifndef ...
        _URB_HEADER                                UrbHeader;
#else
        struct _URB_HEADER                         UrbHeader;
#endif
#ifndef ...
        _URB_SELECT_INTERFACE                      UrbSelectInterface;
#else
        struct _URB_SELECT_INTERFACE              UrbSelectInterface;
#endif
#ifndef ...
        _URB_SELECT_CONFIGURATION                 UrbSelectConfiguration;
#else
        struct _URB_SELECT_CONFIGURATION          UrbSelectConfiguration;
#endif
#ifndef ...
        _URB_PIPE_REQUEST                          UrbPipeRequest;
#else
        struct _URB_PIPE_REQUEST                  UrbPipeRequest;
#endif
#ifndef ...
        _URB_FRAME_LENGTH_CONTROL                UrbFrameLengthControl;
#else
        struct _URB_FRAME_LENGTH_CONTROL          UrbFrameLengthControl;
#endif
#ifndef ...
        _URB_GET_FRAME_LENGTH                   UrbGetFrameLength;
#else
        struct _URB_GET_FRAME_LENGTH             UrbGetFrameLength;
#endif
#ifndef ...
        _URB_SET_FRAME_LENGTH                   UrbSetFrameLength;
#else
        struct _URB_SET_FRAME_LENGTH             UrbSetFrameLength;
#endif
#ifndef ...

```

```
    _URB_GET_CURRENT_FRAME_NUMBER
UrbGetCurrentFrameNumber;
#else
    struct _URB_GET_CURRENT_FRAME_NUMBER
UrbGetCurrentFrameNumber;
#endif
#if ...
    _URB_CONTROL_TRANSFER
UrbControlTransfer;
#else
    struct _URB_CONTROL_TRANSFER
UrbControlTransfer;
#endif
#if ...
    _URB_CONTROL_TRANSFER_EX
UrbControlTransferEx;
#else
    struct _URB_CONTROL_TRANSFER_EX
UrbControlTransferEx;
#endif
#if ...
    _URB_BULK_OR_INTERRUPT_TRANSFER
UrbBulkOrInterruptTransfer;
#else
    struct _URB_BULK_OR_INTERRUPT_TRANSFER
UrbBulkOrInterruptTransfer;
#endif
#if ...
    _URB_ISOCH_TRANSFER
UrbIsochronousTransfer;
#else
    struct _URB_ISOCH_TRANSFER
UrbIsochronousTransfer;
#endif
#if ...
    _URB_CONTROL_DESCRIPTOR_REQUEST
UrbControlDescriptorRequest;
#else
    struct _URB_CONTROL_DESCRIPTOR_REQUEST
UrbControlDescriptorRequest;
#endif
#if ...
    _URB_CONTROL_GET_STATUS_REQUEST
UrbControlGetStatusRequest;
#else
    struct _URB_CONTROL_GET_STATUS_REQUEST
UrbControlGetStatusRequest;
#endif
#if ...
    _URB_CONTROL_FEATURE_REQUEST
UrbControlFeatureRequest;
#else
    struct _URB_CONTROL_FEATURE_REQUEST
UrbControlFeatureRequest;
#endif
#if ...
    _URB_CONTROL_VENDOR_OR_CLASS_REQUEST
UrbControlVendorClassRequest;
#else
    struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST
UrbControlVendorClassRequest;
```

```

#endif
#if ...
    _URB_CONTROL_GET_INTERFACE_REQUEST
UrbControlGetInterfaceRequest;
#else
    struct _URB_CONTROL_GET_INTERFACE_REQUEST
UrbControlGetInterfaceRequest;
#endif
#if ...
    _URB_CONTROL_GET_CONFIGURATION_REQUEST
UrbControlGetConfigurationRequest;
#else
    struct _URB_CONTROL_GET_CONFIGURATION_REQUEST
UrbControlGetConfigurationRequest;
#endif
#if ...
    _URB_OS_FEATURE_DESCRIPTOR_REQUEST
UrbOSFeatureDescriptorRequest;
#else
    struct _URB_OS_FEATURE_DESCRIPTOR_REQUEST
UrbOSFeatureDescriptorRequest;
#endif
#if ...
    _URB_OPEN_STATIC_STREAMS
UrbOpenStaticStreams;
#else
    struct _URB_OPEN_STATIC_STREAMS
UrbOpenStaticStreams;
#endif
#if ...
    _URB_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS
UrbGetIsochPipeTransferPathDelays;
#else
    struct _URB_GET_ISOCH_PIPE_TRANSFER_PATH_DELAYS
UrbGetIsochPipeTransferPathDelays;
#endif
};

} URB, *PURB;

```

## Members

### [UrbHeader](#)

Provides basic information about the request being sent to the host controller driver. For more information, see [\\_URB\\_HEADER](#).

### [UrbSelectInterface](#)

Defines the format of a select interface command for a USB device. For more information, see [\\_URB\\_SELECT\\_INTERFACE](#).

### [UrbSelectConfiguration](#)

Defines the format of a select configuration command for a USB device. For more information, see [\\_URB\\_SELECT\\_CONFIGURATION](#).

#### `UrbPipeRequest`

Defines the format for a command for a pipe in a USB endpoint. For more information, see [\\_URB\\_PIPE\\_REQUEST](#).

#### `UrbFrameLengthControl`

Deprecated in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use.

#### `UrbGetFrameLength`

Deprecated in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use.

#### `UrbSetFrameLength`

Deprecated in Windows 2000 and later operating systems and is not supported by Microsoft. Do not use.

#### `UrbGetCurrentFrameNumber`

Defines the format for a command to get the current frame number on a USB bus. For more information, see [\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#).

#### `UrbControlTransfer`

Defines the format for a command to transmit or receive data on a control pipe. For more information, see [\\_URB\\_CONTROL\\_TRANSFER](#).

#### `UrbControlTransferEx`

Defines the format for a command to transmit or receive data on a control pipe. For more information, see [\\_URB\\_CONTROL\\_TRANSFER\\_EX](#).

Defines the format for a command to transmit or receive data on a control pipe.

#### `UrbBulkOrInterruptTransfer`

Defines the format for a command to transmit or receive data on a bulk pipe, or to receive data from an interrupt pipe. For more information, see [\\_URB\\_BULK\\_OR\\_INTERRUPT\\_TRANSFER](#).

#### `UrbIsochronousTransfer`

Defines the format of an isochronous transfer to a USB device. For more information, see [\\_URB\\_ISOCH\\_TRANSFER](#).

#### `UrbControlDescriptorRequest`

Defines the format for a command to retrieve or set descriptor(s) on a USB device. For more information, see [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#).

#### `UrbControlGetStatusRequest`

Defines the format for a command to get status from a device, interface, or endpoint. For more information, see [\\_URB\\_CONTROL\\_GET\\_STATUS\\_REQUEST](#).

#### `UrbControlFeatureRequest`

Defines the format for a command to set or clear USB-defined features on an device, interface, or endpoint. For more information, see [\\_URB\\_CONTROL\\_FEATURE\\_REQUEST](#).

#### `UrbControlVendorClassRequest`

Defines the format for a command to send or receive a vendor or class-specific request on a device, interface, endpoint, or other device-defined target. For more information, see [\\_URB\\_CONTROL\\_VENDOR\\_OR\\_CLASS\\_REQUEST](#).

#### `UrbControlGetInterfaceRequest`

Defines the format for a command to get the current alternate interface setting for a selected interface. For more information, see [\\_URB\\_CONTROL\\_GET\\_INTERFACE\\_REQUEST](#).

#### `UrbControlGetConfigurationRequest`

Defines the format for a command to get the current configuration for a device. For more information, see [\\_URB\\_CONTROL\\_GET\\_CONFIGURATION\\_REQUEST](#).

#### `UrbOSFeatureDescriptorRequest`

Defines the format for a command to request a Microsoft OS Descriptor. For more information, see [\\_URB\\_OS\\_FEATURE\\_DESCRIPTOR\\_REQUEST](#).

#### `UrbOpenStaticStreams`

Defines the format for a command to open streams in a bulk endpoint of a USB 3.0 device. For more information, see [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#) and [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

## **UrbGetIsochPipeTransferPathDelays**

Defines the format for a command to retrieve delays associated with isochronous transfer programming in the host controller and transfer completion so that the client driver can ensure that the device gets the isochronous packets in time. For more information, see [\\_URB\\_GET\\_ISOCH\\_PIPE\\_TRANSFER\\_PATH\\_DELAYS](#).

## **Remarks**

For information about the function codes to set in each structure, see [\\_URB\\_HEADER](#).

## **Requirements**

<b>Header</b>	usb.h (include Usb.h)

## **See also**

[IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_URB](#)

[USB Structures](#)

# USB\_CONTROLLER\_FLAVOR enumeration (usb.h)

Article02/22/2024

The **USB\_CONTROLLER\_FLAVOR** enumeration specifies the type of USB host controller.

## Syntax

C++

```
typedef enum _USB_CONTROLLER_FLAVOR {
    USB_HcGeneric,
    OHCI_Generic,
    OHCI_Hydra,
    OHCI_NEC,
    UHCI_Generic,
    UHCI_Piix4,
    UHCI_Piix3,
    UHCI_Ich2,
    UHCI_Reserved204,
    UHCI_Ich1,
    UHCI_Ich3m,
    UHCI_Ich4,
    UHCI_Ich5,
    UHCI_Ich6,
    UHCI_Intel,
    UHCI_VIA,
    UHCI_VIA_x01,
    UHCI_VIA_x02,
    UHCI_VIA_x03,
    UHCI_VIA_x04,
    UHCI_VIA_x0E_FIFO,
    EHCI_Generic,
    EHCI_NEC,
    EHCI_Lucent,
    EHCI_NVIDIA_Tegra2,
    EHCI_NVIDIA_Tegra3,
    EHCI_Intel_Medfield
} USB_CONTROLLER_FLAVOR;
```

## Constants

 Expand table

**USB\_HcGeneric**

Indicates a generic host controller.

**OHCI\_Generic**

Indicates a generic OHCI host controller.

**OHCI\_Hydra**

Indicates a Hydra host controller.

**OHCI\_NEC**

Indicates a NEC host controller.

**UHCI\_Generic**

Indicates a generic UHCI host controller.

**UHCI\_Piix4**

Indicates an Intel PIIX4 UHCI host controller.

**UHCI\_Piix3**

Indicates an Intel PIIX3 UHCI host controller.

**UHCI\_Ich2**

Indicates an Intel ICH2 UHCI host controller.

**UHCI\_Reserved204****UHCI\_Ich1**

Indicates an Intel 815 ICH1 UHCI host controller.

**UHCI\_Ich3m**

Indicates an Intel ICH3m UHCI host controller.

**UHCI\_Ich4**

Indicates an Intel ICH4m UHCI host controller.

**UHCI\_Ich5**

Indicates an Intel ICH5m UHCI host controller.

**UHCI\_Ich6**

Indicates an Intel ICH6m UHCI host controller.

**UHCI\_Intel**

Indicates a generic Intel UHCI host controller.

**UHCI\_VIA**

Indicates a generic VIA UHCI host controller.

**UHCI\_VIA\_x01**

Indicates a Revision 1 VIA UHCI host controller.

<code>UHCI_VIA_x02</code>	Indicates a Revision 2 VIA UHCI host controller.
<code>UHCI_VIA_x03</code>	Indicates a Revision 3 VIA UHCI host controller.
<code>UHCI_VIA_x04</code>	Indicates a Revision 4 VIA UHCI host controller.
<code>UHCI_VIA_x0E_FIFO</code>	Indicates a FIFO Revision VIA UHCI host controller.
<code>EHCI_Generic</code>	Indicates a generic EHCI host controller.
<code>EHCI_NEC</code>	Indicates an NEC EHCI host controller.
<code>EHCI_Lucent</code>	Indicates an EHCI Lucent host controller.
<code>EHCI_NVIDIA_Tegra2</code>	Indicates a Revision 2 NVIDIA Tegra EHCI host controller.
<code>EHCI_NVIDIA_Tegra3</code>	Indicates a Revision 3 NVIDIA Tegra EHCI host controller.
<code>EHCI_Intel_Medfield</code>	Indicates an Intel Medfield host controller.

## Requirements

[Expand table](#)

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[USB Constants and Enumerations](#)

[USB\\_CONTROLLER\\_INFO\\_0](#)

# USBD\_ENDPOINT\_OFFLOAD\_INFORMATION structure (usb.h)

Article05/22/2024

ⓘ AI-assisted content. This article was partially created with the help of AI. An author reviewed and revised the content as needed. [Learn more](#)

Stores xHCI-specific V2 information that is used by client drivers to transfer data to and from the offloaded endpoints.

## Syntax

C++

```
typedef struct _USBD_ENDPOINT_OFFLOAD_INFORMATION {
    ULONG             Size;
    USHORT            EndpointAddress;
    ULONG             ResourceId;
    USBD_ENDPOINT_OFFLOAD_MODE Mode;
    ULONG             RootHubPortNumber : 8;
    ULONG             RouteString : 20;
    ULONG             Speed : 4;
    ULONG             UsbDeviceAddress : 8;
    ULONG             SlotId : 8;
    ULONG             MultiTT : 1;
    ULONG             LSOrFSDeviceConnectedToTTHub : 1;
    ULONG             Reserved0 : 14;
    PHYSICAL_ADDRESS TransferSegmentLA;
    PVOID             TransferSegmentVA;
    size_t             TransferRingSize;
    TransferRingInitialCycleBit;
    ULONG             MessageNumber;
    PHYSICAL_ADDRESS EventRingSegmentLA;
    PVOID             EventRingSegmentVA;
    size_t             EventRingSize;
    EventRingInitialCycleBit;
    PHYSICAL_ADDRESS ClientTransferRingSegmentPAIn;
    size_t             ClientTransferRingSizeIn;
    PHYSICAL_ADDRESS ClientDataBufferPAIn;
    size_t             ClientDataBufferSizeIn;
    PHYSICAL_ADDRESS ClientDataBufferLAOut;
    PVOID             ClientDataBufferVAOut;
} USBD_ENDPOINT_OFFLOAD_INFORMATION, *PUSBD_ENDPOINT_OFFLOAD_INFORMATION,
USBD_ENDPOINT_OFFLOAD_INFORMATION_V2,
*PUSBD_ENDPOINT_OFFLOAD_INFORMATION_V2;
```

# Members

`Size`

The size of this structure.

`EndpointAddress`

Specifies the USB-defined endpoint address.

`ResourceId`

The resource identifier.

`Mode`

A [USBD\\_ENDPOINT\\_OFFLOAD\\_MODE](#) value that indicates whether endpoint offloading is handled in software or the USB device or host controller.

`RootHubPortNumber`

The port number of the root hub to which the device is connected.

`RouteString`

The route string describing the path from the root hub to the device.

`Speed`

The speed of the USB device.

`UsbDeviceAddress`

The USB address of the device.

`SlotId`

The slot ID of the device.

`MultiTT`

Indicates if the device is connected to a hub with transaction translators.

`LSOrFSDeviceConnectedToTTHub`

Indicates if the device is a low-speed or full-speed device connected to a hub with transaction translators.

`Reserved0`

Reserved.

`TransferSegmentLA`

The physical address of the transfer ring segment.

`TransferSegmentVA`

The virtual address of the transfer ring segment.

`TransferRingSize`

The size of the transfer ring.

`TransferRingInitialCycleBit`

The initial cycle bit of the transfer ring.

`MessageNumber`

The message number for the secondary event ring.

`EventRingSegmentLA`

The physical address of the event ring segment.

`EventRingSegmentVA`

The virtual address of the event ring segment.

`EventRingSize`

The size of the event ring, in bytes.

`EventRingInitialCycleBit`

The initial cycle bit of the event ring.

`ClientTransferRingSegmentPAIn`

The physical address of the client-provided transfer ring segment.

`ClientTransferRingSizeIn`

The size of the client-provided transfer ring segment.

`ClientDataBufferPAIn`

The physical address of the client-provided data buffer.

`ClientDataBufferSizeIn`

The size of the client-provided data buffer.

`ClientDataBufferLAOut`

The physical address of the mapped data buffer.

`ClientDataBufferVAOut`

The virtual address of the mapped data buffer.

## remarks

This structure duplicates and extends [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION\\_V1](#).

This structure supports two versions, indicated by `USBD_ENDPOINT_OFFLOAD_INFORMATION` and `USBD_ENDPOINT_OFFLOAD_INFORMATION_V2`, with the latter possibly including additional fields beyond what is documented here. The structure facilitates detailed configuration and management of USB endpoint offload.

## see-also

- [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION\\_V1](#)

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usb.h

## Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

# USBD\_ENDPOINT\_OFFLOAD\_INFORMATION\_V1 structure (usb.h)

Article05/22/2024

ⓘ AI-assisted content. This article was partially created with the help of AI. An author reviewed and revised the content as needed. [Learn more](#)

Stores xHCI-specific V1 information that is used by client drivers to transfer data to and from the offloaded endpoints.

## Syntax

C++

```
typedef struct _USBD_ENDPOINT_OFFLOAD_INFORMATION_V1 {
    ULONG             Size;
    USHORT            EndpointAddress;
    ULONG             ResourceId;
    USBD_ENDPOINT_OFFLOAD_MODE Mode;
    ULONG             RootHubPortNumber : 8;
    ULONG             RouteString : 20;
    ULONG             Speed : 4;
    ULONG             UsbDeviceAddress : 8;
    ULONG             SlotId : 8;
    ULONG             MultiTT : 1;
    ULONG             LSOrFSDeviceConnectedToTTHub : 1;
    ULONG             Reserved0 : 14;
    PHYSICAL_ADDRESS TransferSegmentLA;
    PVOID             TransferSegmentVA;
    size_t            TransferRingSize;
    ULONG             TransferRingInitialCycleBit;
    ULONG             MessageNumber;
    PHYSICAL_ADDRESS EventRingSegmentLA;
    PVOID             EventRingSegmentVA;
    size_t            EventRingSize;
    ULONG             EventRingInitialCycleBit;
} USBD_ENDPOINT_OFFLOAD_INFORMATION_V1,
*PUSBD_ENDPOINT_OFFLOAD_INFORMATION_V1;
```

## Members

Size

The size of this structure.

### `EndpointAddress`

Specifies the USB-defined endpoint address.

### `ResourceId`

The resource identifier.

### `Mode`

A [USBD\\_ENDPOINT\\_OFFLOAD\\_MODE](#) value that indicates whether endpoint offloading is handled in software or the USB device or host controller.

### `RootHubPortNumber`

The port number of the root hub to which the device is connected.

### `RouteString`

The route string describing the path from the root hub to the device.

### `Speed`

The speed of the USB device.

### `UsbDeviceAddress`

The USB address of the device.

### `SlotId`

The slot ID of the device.

### `MultiTT`

Indicates if the device is connected to a hub with transaction translators.

### `LS0rFSDeviceConnectedToTTHub`

Indicates if the device is a low-speed or full-speed device connected to a hub with transaction translators.

### `Reserved0`

Reserved.

### `TransferSegmentLA`

The physical address of the transfer ring segment.

`TransferSegmentVA`

The virtual address of the transfer ring segment.

`TransferRingSize`

The size of the transfer ring.

`TransferRingInitialCycleBit`

The initial cycle bit of the transfer ring.

`MessageNumber`

The message number for the secondary event ring.

`EventRingSegmentLA`

The physical address of the event ring segment.

`EventRingSegmentVA`

The virtual address of the event ring segment.

`EventRingSize`

The size of the event ring, in bytes.

`EventRingInitialCycleBit`

The initial cycle bit of the event ring.

## see-also

- [USBD\\_ENDPOINT\\_OFFLOAD\\_INFORMATION](#)

# Requirements

  Expand table

Requirement	Value
Header	usb.h

---

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USBD\_ENDPOINT\_OFFLOAD\_MODE enumeration (usb.h)

Article02/22/2024

Defines values for endpoint offloading options in the USB device or host controller.

## Syntax

C++

```
typedef enum _USBD_ENDPOINT_OFFLOAD_MODE {
    UsbdEndpointOffloadModeNotSupported,
    UsbdEndpointOffloadSoftwareAssisted,
    UsbdEndpointOffloadHardwareAssisted
} USBD_ENDPOINT_OFFLOAD_MODE;
```

## Constants

[+] Expand table

<code>UsbdEndpointOffloadModeNotSupported</code> Endpoint offloading is not supported.
<code>UsbdEndpointOffloadSoftwareAssisted</code> Endpoint offloading is handled by the software.
<code>UsbdEndpointOffloadHardwareAssisted</code> Endpoint offloading is handled in the USB device or host controller hardware.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016

Requirement	Value
Header	usb.h

## See also

[USBD\\_QueryUsbCapability](#)

# USBD\_INTERFACE\_INFORMATION structure (usb.h)

Article02/22/2024

The **USBD\_INTERFACE\_INFORMATION** structure holds information about an interface for a configuration on a USB device.

## Syntax

C++

```
typedef struct _USBD_INTERFACE_INFORMATION {
    USHORT          Length;
    UCHAR           InterfaceNumber;
    UCHAR           AlternateSetting;
    UCHAR           Class;
    UCHAR           SubClass;
    UCHAR           Protocol;
    UCHAR           Reserved;
    USBD_INTERFACE_HANDLE InterfaceHandle;
    ULONG           NumberOfPipes;
    USBD_PIPE_INFORMATION Pipes[1];
} USBD_INTERFACE_INFORMATION, *PUSBD_INTERFACE_INFORMATION;
```

## Members

**Length**

Specifies the length, in bytes, of this structure.

**InterfaceNumber**

Specifies the device-defined index identifier for this interface.

**AlternateSetting**

Specifies a device-defined index identifier that indicates which alternate setting this interface is using, should use, or describes.

**Class**

Contains a USB-assigned identifier that specifies a USB-defined class that this interface conforms to.

### SubClass

Contains a USB-assigned identifier that specifies a USB-defined subclass that this interface conforms to. This code is specific to the code in **Class**.

### Protocol

Contains a USB-assigned identifier that specifies a USB-defined protocol that this interface conforms to. This code is specific to the codes in **Class** and **SubClass**.

### Reserved

Reserved.

### InterfaceHandle

Contains a host controller driver-defined handle that is used to access this interface. This member should be treated as opaque.

### NumberOfPipes

Specifies the number of pipes (endpoints) in this interface.

### Pipes[1]

Pointer to the first element in the array of [USBD\\_PIPE\\_INFORMATION](#) structures. The length of the array depends on the number of endpoints in the interface descriptor.

## Remarks

Members that are part of this structure, but not described here, should be treated as opaque and considered to be reserved for system use.

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[USB Structures](#)

[USBD\\_PIPE\\_INFORMATION](#)

# USBD\_ISO\_PACKET\_DESCRIPTOR structure (usb.h)

Article04/01/2021

The **USBD\_ISO\_PACKET\_DESCRIPTOR** structure is used by USB client drivers to describe an isochronous transfer packet.

## Syntax

C++

```
typedef struct _USBD_ISO_PACKET_DESCRIPTOR {
    ULONG      Offset;
    ULONG      Length;
    USBD_STATUS Status;
} USBD_ISO_PACKET_DESCRIPTOR, *PUSBD_ISO_PACKET_DESCRIPTOR;
```

## Members

**Offset**

Specifies the offset, in bytes, of the buffer for this packet from the beginning of the entire isochronous transfer buffer.

**Length**

Set by the host controller to indicate the actual number of bytes received from the device for isochronous IN transfers. **Length** not used for isochronous OUT transfers.

**Status**

Contains the status, on return from the host controller driver, of this transfer packet.

## Remarks

This structure is used as part of an isochronous transfer request to the host controller driver using the [\\_URB\\_ISOCH\\_TRANSFER](#) structure. The **Offset** member contains the offset from the beginning of the **TransferBuffer** or **TransferBufferMDL** members of [\\_URB\\_ISOCH\\_TRANSFER](#).

# Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[How to Transfer Data to USB Isochronous Endpoints](#)

[USB Structures](#)

[USBD\\_IsochUrbAllocate](#)

[\\_URB\\_ISOCH\\_TRANSFER](#)

# USBD\_PIPE\_INFORMATION structure (usb.h)

Article04/01/2021

The **USBD\_PIPE\_INFORMATION** structure is used by USB client drivers to hold information about a pipe from a specific interface.

## Syntax

C++

```
typedef struct _USBD_PIPE_INFORMATION {
    USHORT          MaximumPacketSize;
    UCHAR           EndpointAddress;
    UCHAR           Interval;
    USBD_PIPE_TYPE PipeType;
    USBD_PIPE_HANDLE PipeHandle;
    ULONG           MaximumTransferSize;
    ULONG           PipeFlags;
} USBD_PIPE_INFORMATION, *PUSBD_PIPE_INFORMATION;
```

## Members

### MaximumPacketSize

Specifies the maximum packet size, in bytes, that this pipe handles. This value must be less than or equal to the value of **wMaxPacketSize** in the endpoint descriptor. The USB stack ignores this value if the **USBD\_PFF\_CHANGE\_MAX\_PACKET** flag is not set in the **PipeFlags** member.

For high-speed isochronous endpoints, the received **MaximumPacketSize** value includes the number of bytes that can be transferred in additional transactions, if the endpoint supports them. For more information, see Remarks.

### EndpointAddress

Specifies the bus address for this pipe.

### Interval

Contains the polling interval, indicated by the **bInterval** field in the corresponding endpoint descriptor ([USB\\_ENDPOINT\\_DESCRIPTOR](#)). This value is only valid for interrupt

and isochronous pipes. For other types of pipe, this value should be ignored. It reflects the device's configuration in firmware. Drivers cannot change it.

The polling interval, together with the speed of the device and the type of host controller, determine the frequency with which the driver should initiate a transfer. The value in **Interval** does not represent a fixed amount of time. It is a relative value, and the actual polling frequency will also depend on whether the device and the USB host controller operate at low, full or high speed.

If either the host controller or the device operates at low speed, the period of time between transfers (also known as the "polling period") is measured in units of 1 millisecond frames, and the period is related to the value in **Interval** as indicated the following table:

<b>Interval</b>	<b>Polling Period (1-millisecond frames)</b>	<b>Interrupt</b>	<b>Isochronous</b>
0 to 15	8	Supported.	Not supported.
16 to 35	16	Supported.	Not supported.
36 to 255	32	Supported.	Not supported.

For devices and host controllers that can operate at full speed, the period is measured in units of 1 millisecond frames. For full-speed isochronous transfers, the **Interval** value and the polling period is always 1. That value indicates that data can be transferred in every frame. For full-speed interrupt transfers, the polling period is derived from the **Interval** value. The following table indicates the supported values for interrupt and isochronous endpoints.

<b>Interval</b>	<b>Polling Period (1-millisecond frames)</b>	<b>Interrupt</b>	<b>Isochronous</b>
1	1	Supported.	Supported.
2 to 3	2	Supported.	Not supported.
4 to 7	4	Supported.	Not supported.
8 to 15	8	Supported.	Not supported.
16 to 31	16	Supported.	Not supported.
32 to 255	32	Supported.	Not supported.

For devices and host controllers that can operate at high speed, the period is measured in units of microframes. The polling period is derived from the **Interval** value by using the formula `Polling period = 2 ** (Interval - 1)`. The calculated values are indicated in the following table:

<b>Interval</b>	<b>Polling Period (microframes)</b>	<b>Interrupt</b>	<b>Isochronous</b>
1	1	Supported.	Supported.
2	2	Supported.	Supported.
3	4	Supported.	Supported.
4	8	Supported.	Supported.
5	16	Supported.	Not supported.
6 to 255	32	Supported.	Not supported.

The supported polling periods for high-speed isochronous transfers are 1, 2, 4, and 8. If a client driver submits a URB\_FUNCTION\_ISOCH\_TRANSFER request for a high speed isochronous endpoint with polling period greater than 8, the request fails with status USBD\_STATUS\_INVALID\_PARAMETER. For information about isochronous transfers, see [How to Transfer Data to USB Isochronous Endpoints](#).

The mappings in the preceding tables between periods and polling intervals are valid in Microsoft Windows 2000 and later versions of the Windows operating system.

#### `PipeType`

Specifies what type of transfers this pipe uses. These values are defined in the [USBD\\_PIPE\\_TYPE](#) enumeration.

#### `PipeHandle`

Specifies an opaque handle to the bulk or interrupt pipe. The host controller driver returns this handle when the client driver selects the device configuration with a URB of type URB\_FUNCTION\_SELECT\_CONFIGURATION or when the client driver changes the settings for an interface with a URB of type URB\_FUNCTION\_SELECT\_INTERFACE.

#### `MaximumTransferSize`

Specifies the maximum size, in bytes, for a transfer request on this pipe. In Windows Server 2003, Windows XP and later operating systems, this member is not used and does not contain valid data.

For information about the maximum transfer sizes of each type of USB endpoint in different versions of Windows, see [USB Transfer and Packet Sizes](#).

**Note** For WinUSB, do not use **MaximumTransferSize** to determine the maximum size of a USB transfer. Instead, use the **MAXIMUM\_TRANSFER\_SIZE** value retrieved by **WinUsb\_GetPipePolicy**.

#### PipeFlags

Contains a bitwise-OR of pipe flags that the driver can use to specify certain configurable characteristics of the pipe. The driver specifies these pipe characteristics when it selects the configuration of a USB device with a URB request whose function type is **URB\_FUNCTION\_SELECT\_CONFIGURATION**.

The following table explains the meaning of each pipe flag:

Flag name	Meaning
<b>USBD_PU_CHANGE_MAX_PACKET</b>	Indicates that the driver is overriding the endpoint maximum packet size with the value specified in <b>MaximumPacketSize</b> . This value must be less than or equal to the default maximum specified in the pipe's endpoint descriptor.

## Remarks

This structure contains information for an endpoint, retrieved from the device's interface descriptor. For an explanation of how to obtain the information in **USBD\_PIPE\_INFORMATION** from the interface descriptor, see [How to Select a Configuration for a USB Device](#).

The **MaximumPacketSize** value is derived from the first 11 bits of the **wMaxPacketSize** field of the endpoint descriptor, which indicates the maximum number of bytes that the host controller can send to or receive from the endpoint in a single transaction.

Typically, for high-speed transfers, the host controller sends or receives one transaction per microframe. However, high speed, high bandwidth isochronous or interrupt endpoints support higher data rates through additional transactions. This allows the host controller to transfer up to 3072 bytes in a single microframe. The number of additional transactions supported by that type of endpoint is indicated by bits 12..11 of **wMaxPacketSize** (least significant bit is 0). 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 value received in **MaximumPacketSize** for an isochronous endpoint (high-speed and high-bandwidth) indicates the total number of bytes that the host controller can send to or receive from the endpoint in one microframe. The value includes the number of bytes in additional transactions, if the endpoint supports them. For example, consider the following isochronous endpoint characteristics:

- **wMaxPacketSize** is 1,024
- Bits 12..11 indicate 2
- **Interval** is 1.

In the preceding example, the value received in **MaximumPacketSize** is 3,072 bytes (Total transactions \* **wMaxPacketSize**). Because **Interval** is 1, the polling period is 1. Thus, the host controller can transfer 3,072 bytes in each microframe of a frame. In a single I/O request (described in one URB), the host controller can transfer no more than 24,576 bytes.

## Requirements

Header	usb.h (include Usb.h)
--------	-----------------------

## See also

[USB Structures](#)

[USB\\_ENDPOINT\\_DESCRIPTOR](#)

# USBD\_PIPE\_TYPE enumeration (usb.h)

Article04/16/2024

The **USBD\_PIPE\_TYPE** enumerator indicates the type of pipe.

## Syntax

C++

```
typedef enum _USBD_PIPE_TYPE {
    UsbdPipeTypeControl,
    UsbdPipeTypeIsochronous,
    UsbdPipeTypeBulk,
    UsbdPipeTypeInterrupt
} USBD_PIPE_TYPE;
```

## Constants

[ ] [Expand table](#)

	<b>UsbdPipeTypeControl</b>
	Indicates that the pipe is a control pipe.
	<b>UsbdPipeTypeIsochronous</b>
	Indicates that the pipe is an isochronous transfer pipe.
	<b>UsbdPipeTypeBulk</b>
	Indicates that the pipe is a bulk transfer pipe.
	<b>UsbdPipeTypeInterrupt</b>
	Indicates that the pipe is an interrupt pipe.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usb.h (include Usb.h)

## See also

[USB Constants and Enumerations](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USBD\_STREAM\_INFORMATION structure (usb.h)

Article02/22/2024

The **USBD\_STREAM\_INFORMATION** structure stores information about a stream associated with a bulk endpoint.

## Syntax

C++

```
typedef struct _USBD_STREAM_INFORMATION {
    USBD_PIPE_HANDLE PipeHandle;
    ULONG             StreamID;
    ULONG             MaximumTransferSize;
    ULONG             PipeFlags;
} USBD_STREAM_INFORMATION, *PUSBD_STREAM_INFORMATION;
```

## Members

PipeHandle

An opaque handle to the stream.

StreamID

Stream identifier. The open-static streams request obtains stream identifiers that are assigned by the USB driver stack.

MaximumTransferSize

Maximum transfer size (in bytes) that a client driver can send in a single URB for an I/O transfer to the stream.

PipeFlags

Reserved. Do not use.

## Remarks

A client driver allocates an array of **USBD\_STREAM\_INFORMATION** structures and sends it in an open-streams request (URB\_FUNCTION\_OPEN\_STATIC\_STREAMS). Upon completion, the USB driver stack retrieves stream information and populates each **USBD\_STREAM\_INFORMATION** structure with stream information. The stream identifiers returned by the request are sequential and start at 1.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 8
Header	usb.h

## See also

[How to Open and Close Static Streams in a USB Bulk Endpoint](#)

[URB](#)

[USB Structures](#)

[\\_URB\\_HEADER](#)

[\\_URB\\_OPEN\\_STATIC\\_STREAMS](#)

# USBD\_VERSION\_INFORMATION structure (usb.h)

Article02/22/2024

The **USBD\_VERSION\_INFORMATION** structure is used by the [GetUSBDIVersion](#) function to report its output data.

## Syntax

C++

```
typedef struct _USBD_VERSION_INFORMATION {
    ULONG USBDI_Version;
    ULONG Supported_USB_Version;
} USBD_VERSION_INFORMATION, *PUSBD_VERSION_INFORMATION;
```

## Members

### USBDI\_Version

Contains a binary-coded decimal USB interface version number. Released interface versions are listed in the following table.

[+] Expand table

Operating system	Interface version
Windows 98 Gold	0x00000102
Windows 98 SE	0x00000200
Windows 2000	0x00000300
Windows Millennium Edition	0x00000400
Windows XP	0x00000500
Windows Vista	0x00000600
Windows 7	
Windows 8	

### Supported\_USB\_Version

Contains a binary-coded decimal USB specification version number.

## Remarks

[GetUSBDIVersion](#) is deprecated in Windows 8 and later versions of the operating system. To determine whether a particular version is supported by the underlying USB driver stack, the client driver must call [USBD\\_IsInterfaceVersionSupported](#).

## Requirements

[+] Expand table

Requirement	Value
Header	usb.h (include Usbbusif.h)

## See also

[GetUSBDIVersion](#)

[USB Bus Driver Interface \(USBDI\) Routines](#)

[USB Structures](#)

# usbbusif.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usbbusif.h contains the following programming interfaces:

## Callback functions

### [PUSB\\_BUSIFFN\\_ENUM\\_LOG\\_ENTRY](#)

This callback function is not supported. The EnumLogEntry routine makes a log entry.

### [PUSB\\_BUSIFFN\\_GETUSBDI\\_VERSION](#)

The GetUSBDIVersion routine returns the USB interface version number and the version number of the USB specification that defines the interface, along with information about host controller capabilities.

### [PUSB\\_BUSIFFN\\_IS\\_DEVICE\\_HIGH\\_SPEED](#)

The USB\_BUSIFFN\_IS\_DEVICE\_HIGH\_SPEED routine returns TRUE if the device is operating at high speed.

### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_INFORMATION](#)

The QueryBusInformation routine gets information about the bus.

### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_TIME](#)

The QueryBusTime function gets the current 32-bit USB frame number.

### [PUSB\\_BUSIFFN\\_QUERY\\_BUS\\_TIME\\_EX](#)

The QueryBusTimeEx routine gets the current 32-bit USB micro-frame number.

### [PUSB\\_BUSIFFN\\_QUERY\\_CONTROLLER\\_TYPE](#)

The QueryControllerType routine gets information about the USB host controller to which the USB device is attached.

### [PUSB\\_BUSIFFN\\_SUBMIT\\_ISO\\_OUT\\_URB](#)

This callback function is not supported. The SubmitIsoOutUrb function submits a USB request block (URB) directly to the bus driver without requiring the allocation of an IRP.

#### [USBC\\_START\\_DEVICE\\_CALLBACK](#)

The USBC\_START\_DEVICE\_CALLBACK routine allows a USB client driver to provide a custom definition of the interface collections on a device.

## Structures

#### [USB\\_BUS\\_INFORMATION\\_LEVEL\\_0](#)

The USB\_BUS\_INFORMATION\_LEVEL\_0 structure is used in conjunction with the QueryBusInformation interface routine to report information about the bus.

#### [USB\\_BUS\\_INFORMATION\\_LEVEL\\_1](#)

The USB\_BUS\_INFORMATION\_LEVEL\_1 structure is used in conjunction with the QueryBusInformation interface routine to report information about the bus.

#### [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#)

The USB\_BUS\_INTERFACE\_USBDI\_V0 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

#### [USB\\_BUS\\_INTERFACE\\_USBDI\\_V1](#)

The USB\_BUS\_INTERFACE\_USBDI\_V1 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

#### [USB\\_BUS\\_INTERFACE\\_USBDI\\_V2](#)

The USB\_BUS\_INTERFACE\_USBDI\_V2 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

#### [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#)

The USB\_BUS\_INTERFACE\_USBDI\_V3 structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

#### [USBC\\_DEVICE\\_CONFIGURATION\\_INTERFACE\\_V1](#)

The USBC\_DEVICE\_CONFIGURATION\_INTERFACE\_V1 structure is exposed by the vendor-supplied filter drivers to assist the USB generic parent driver in defining interface collections.

## [USBC\\_FUNCTION\\_DESCRIPTOR](#)

The USBC\_FUNCTION\_DESCRIPTOR structure describes a USB function and its associated interface collection.

# PUSB\_BUSIFFN\_ENUM\_LOG\_ENTRY callback function (usbbusif.h)

Article02/22/2024

This callback function is not supported.

The *EnumLogEntry* routine makes a log entry.

## Syntax

C++

```
typedef NTSTATUS  
(USB_BUSIFFN *PUSB_BUSIFFN_ENUM_LOG_ENTRY) (  
    IN PVOID,  
    IN ULONG,  
    IN ULONG,  
    IN ULONG,  
    IN ULONG  
);
```

## Parameters

[in] unnamedParam1

Handle to the bus context returned in the *BusContext* member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V2](#) structure by an IRP\_MN\_QUERY\_INTERFACE request.

[in] unnamedParam2

Vendor-defined data to store in the enumeration log.

[in] unnamedParam3

Vendor-defined data to store in the enumeration log.

[in] unnamedParam4

Vendor-defined data to store in the enumeration log.

[in] unnamedParam5

Vendor-defined data to store in the enumeration log.

# Return value

The *EnumLogEntry* routine always returns STATUS\_SUCCESS.

## Remarks

The routine definition that is provided in the example is a routine whose parameters are just placeholder names. The actual prototype of the routine is declared in *usbbusif.h*.

## Requirements

 Expand table

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	ANY

# PUSB\_BUSIFFN\_GETUSBDI\_VERSION callback function (usbbusif.h)

Article02/22/2024

The **GetUSBDIVersion** routine returns the USB interface version number and the version number of the USB specification that defines the interface, along with information about host controller capabilities.

## ⓘ Note

**USBD\_IsInterfaceVersionSupported** replaces the **GetUSBDIVersion** routine. To determine the capabilities of the host controller and the underlying USB driver stack, call **USBD\_QueryUsbCapability**.

## Syntax

C++

```
typedef VOID
(USB_BUSIFFN *PUSB_BUSIFFN_GETUSBDI_VERSION) (
    IN PVOID,
    IN OUT PUSBD_VERSION_INFORMATION,
    IN OUT PULONG
);
```

## Parameters

[in] unnamedParam1

Handle returned in the **BusContext** member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#) structure by an **IRP\_MN\_QUERY\_INTERFACE** request.

[out, optional] unnamedParam2

Returns the host capability flags. Currently, no host capability flags are reported.

[out, optional] unnamedParam3

Returns a pointer to a [USBD\\_VERSION\\_INFORMATION](#) structure that contains the USB interface version number and the USB specification version number.

# Return value

None

## Remarks

The function returns the highest USBDI interface version supported by the port driver. This function replaces the [USBD\\_GetUSBDIVersion](#) library function provided by *usbd.sys*.

## Requirements

 Expand table

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USBD\\_GetUSBDIVersion](#)
- [USBD\\_VERSION\\_INFORMATION](#)
- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#)

# PUSB\_BUSIFFN\_IS\_DEVICE\_HIGH\_SPEED callback function (usbbusif.h)

Article 02/22/2024

The USB\_BUSIFFN\_IS\_DEVICE\_HIGH\_SPEED routine returns `TRUE` if the device is operating at high speed.

## Syntax

C++

```
typedef BOOLEAN  
    (USB_BUSIFFN *PUSB_BUSIFFN_IS_DEVICE_HIGH_SPEED) (  
        IN PVOID  
    );
```

## Parameters

`[in, optional] unnamedParam1`

Handle returned in the `BusContext` member of the `USB_BUS_INTERFACE_USBDI_V1` structure by an `IRP_MN_QUERY_INTERFACE` request.

## Return value

`USB_BUSIFFN_IS_DEVICE_HIGH_SPEED` implementation returns `TRUE` if the USB device is operating at high speed USB 2.0 compliant device; `FALSE` otherwise.

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V1](#)

# PUSB\_BUSIFFN\_QUERY\_BUS\_INFORMATION callback function (usbbusif.h)

Article02/05/2022

The **QueryBusInformation** routine gets information about the bus.

## Syntax

C++

```
typedef NTSTATUS  
    (USB_BUSIFFN *PUSB_BUSIFFN_QUERY_BUS_INFORMATION) (  
        IN PVOID,  
        IN ULONG,  
        IN OUT PVOID,  
        IN OUT PULONG,  
        OUT PULONG  
    );
```

## Parameters

[in] unnamedParam1

Handle returned in the **BusContext** member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#) structure by an IRP\_MN\_QUERY\_INTERFACE request.

[out, optional] unnamedParam2

Specifies the length of the output data.

[in, out] unnamedParam3

Pointer to a buffer that receives the requested bus information.

[out] unnamedParam4

On input, the length of the buffer specified by *BusInformationBuffer*. On output, the length of the output data.

[in] unnamedParam5

Specifies the level of information to be returned. If *Level* is 0, the function returns the total bandwidth and the total consumed bandwidth in bits per second. If *Level* is 1, the

function returns the symbolic name of the controller in Unicode, in addition to the total bandwidth and the total consumed bandwidth.

## Return value

**QueryBusInformation** returns one of the following values:

Return code	Description
<code>STATUS_SUCCESS</code>	The call completed successfully.
<code>STATUS_BUFFER_TOO_SMALL</code>	<p>The buffer was too small. This error code is returned in two cases:</p> <p>Whenever <i>Level</i> == 0, this error code is returned if the size of the buffer pointed to by <i>BusInformationBuffer</i> is less than the size of the <a href="#">USB_BUS_INFORMATION_LEVEL_0</a> structure.</p> <p>Whenever Level == 1, this error code is returned if the size of the buffer pointed to by <i>BusInformationBuffer</i> less than the size of the <a href="#">USB_BUS_INFORMATION_LEVEL_1</a> structure.</p>

## Remarks

The exact information returned by this routine depends on the value of the *Level* parameter. This routine replaces the **USBD\_QueryBusInformation** library function provided by usbd.sys.

## Requirements

Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USB\\_BUS\\_INFORMATION\\_LEVEL\\_0](#)
- [USB\\_BUS\\_INFORMATION\\_LEVEL\\_1](#)
- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#)

# PUSB\_BUSIFFN\_QUERY\_BUS\_TIME callback function (usbbusif.h)

Article02/22/2024

The *QueryBusTime* function gets the current 32-bit USB frame number.

## Syntax

C++

```
typedef NTSTATUS
(USB_BUSIFFN *PUSB_BUSIFFN_QUERY_BUS_TIME) (
    IN PVOID,
    IN PULONG
);
```

## Parameters

[in] unnamedParam1

Handle returned in the **BusContext** member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#) structure by an IRP\_MN\_QUERY\_INTERFACE request.

[out, optional] unnamedParam2

Receives the current USB frame number.

## Return value

Returns STATUS\_SUCCESS on success, and the appropriate error code on failure.

## Remarks

This routine replaces the **USBD\_QueryBusTime** library function provided by *usbd.sys*.

## Requirements

[+] Expand table

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#)

# PUSB\_BUSIFFN\_QUERY\_BUS\_TIME\_EX

## callback function (usbbusif.h)

Article 02/22/2024

This request is not supported.

The **QueryBusTimeEx** routine gets the current 32-bit USB micro-frame number.

## Syntax

C++

```
typedef NTSTATUS  
    (USB_BUSIFFN *PUSB_BUSIFFN_QUERY_BUS_TIME_EX) (  
        IN PVOID,  
        IN PULONG  
    );
```

## Parameters

[in] unnamedParam1

Handle returned in the **BusContext** member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#) structure by an **IRP\_MN\_QUERY\_INTERFACE** request.

[out] unnamedParam2

Receives the current USB micro-frame number.

## Return value

**QueryBusTimeEx** returns one of the following values:

[ ] Expand table

Return code	Description
<code>STATUS_SUCCESS</code>	The call completed successfully.
<code>STATUS_NOT_SUPPORTED</code>	The function was called for a USB host controller that does not support USB 2.0.

## Remarks

`QueryBusTimeEx` gets the current USB 2.0 frame/micro-frame number when called for a USB device attached to a USB 2.0 host controller.

The lowest 3 bits of the returned micro-frame value will contain the current 125us micro-frame, while the upper 29 bits will contain the current 1ms USB frame number.

## Requirements

[] [Expand table](#)

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#)

# PUSB\_BUSIFFN\_QUERY\_CONTROLLER\_T YPE callback function (usbbusif.h)

Article02/05/2022

The **QueryControllerType** routine gets information about the USB host controller to which the USB device is attached.

## Syntax

C++

```
typedef NTSTATUS  
    (USB_BUSIFFN *PUSB_BUSIFFN_QUERY_CONTROLLER_TYPE) (  
        IN PVOID,  
        OUT PULONG,  
        OUT PUSHORT,  
        OUT PUSHORT,  
        OUT P UCHAR,  
        OUT P UCHAR,  
        OUT P UCHAR,  
        OUT P UCHAR  
    );
```

## Parameters

[in] unnamedParam1

Handle returned in the **BusContext** member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#) structure by an IRP\_MN\_QUERY\_INTERFACE request.

[out] unnamedParam2

Reserved. Do not use.

[out] unnamedParam3

Pointer to a UCHAR variable that receives the PCI class for the USB host controller.

[out] unnamedParam4

Pointer to a USHORT variable that receives the PCI device ID for the USB host controller.

[out] unnamedParam5

Pointer to a UCHAR variable that receives the PCI programming interface for the USB host controller.

[out] unnamedParam6

Pointer to a UCHAR variable that receives the PCI revision number for the USB host controller.

[out] unnamedParam7

Pointer to a UCHAR variable that receives the PCI subclass for the USB host controller.

[out] unnamedParam8

Pointer to a USHORT variable that receives the PCI vendor ID for the USB host controller.

## Return value

Returns STATUS\_SUCCESS on success, and the appropriate error code on failure.

## Remarks

*PciClass* is typically set to PCI\_CLASS\_SERIAL\_BUS\_CTRLR (0x0C).

*PciSubClass* is typically set to PCI\_SUBCLASS\_SB\_USB (0x03).

*PciProgif* is typically set to one of the following values:

- 0x00 - Universal Host Controller Interface (UHCI)
- 0x10 - Open Host Controller Interface (OHCI)
- 0x20 - Enhanced Host Controller Interface (EHCI)

## Requirements

Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	< = DISPATCH_LEVEL

## See also

- [USB\\_BUS\\_INTERFACE\\_USBDI\\_V3](#)

# PUSB\_BUSIFFN\_SUBMIT\_ISO\_OUT\_URB callback function (usbbusif.h)

Article02/22/2024

This callback function is not supported.

The *SubmitIsoOutUrb* function submits a USB request block (URB) directly to the bus driver without requiring the allocation of an IRP.

## Syntax

C++

```
typedef NTSTATUS  
    (USB_BUSIFFN *PUSB_BUSIFFN_SUBMIT_ISO_OUT_URB) (  
        IN PVOID,  
        IN PURB  
    );
```

## Parameters

[in] unnamedParam1

Handle returned in the *BusContext* member of the [USB\\_BUS\\_INTERFACE\\_USBDI\\_V0](#) structure by an *IRP\_MN\_QUERY\_INTERFACE* request.

[in] unnamedParam2

Pointer to the [URB](#) to be passed to the port driver.

## Return value

*SubmitIsoOutUrb* returns one of the following values:

  Expand table

Return code	Description
<a href="#">STATUS_SUCCESS</a>	The call completed successfully.

Return code	Description
STATUS_NOT_SUPPORTED	Fast isochronous interfaces and real-time threads are not supported by the host controller.

## Remarks

This function replaces the **USBD\_BusSubmitIsoOutUrb** library function provided by *usb3.sys*.

This function allows clients running in real-time threads at an elevated IRQL to have rapid access to the bus driver. This USB host controller must support real-time threads for this function to work.

The calling driver forfeits any packet-level error information when calling this function.

## Requirements

[Expand table](#)

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)
IRQL	ANY

# USB\_BUS\_INFORMATION\_LEVEL\_0 structure (usbbusif.h)

Article02/22/2024

The **USB\_BUS\_INFORMATION\_LEVEL\_0** structure is used in conjunction with the [QueryBusInformation](#) interface routine to report information about the bus.

## Syntax

C++

```
typedef struct _USB_BUS_INFORMATION_LEVEL_0 {
    ULONG TotalBandwidth;
    ULONG ConsumedBandwidth;
} USB_BUS_INFORMATION_LEVEL_0, *PUSB_BUS_INFORMATION_LEVEL_0;
```

## Members

TotalBandwidth

Specifies the total bandwidth, in bits per second, available on the bus.

ConsumedBandwidth

Specifies the mean bandwidth that is already in use, in bits per second.

## Remarks

Caller must set the *Level* parameter of the [QueryBusInformation](#) routine to zero.

For more information about how to obtain the proper level of USB interface, see [USB Routines](#).

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Available in Microsoft Windows XP and later operating systems.

Requirement	Value
	Usbbusif.h (include Usbbusif.h)

## See also

[QueryBusInformation](#)

[USB Structures](#)

# USB\_BUS\_INFORMATION\_LEVEL\_1 structure (usbbusif.h)

Article09/01/2022

The **USB\_BUS\_INFORMATION\_LEVEL\_1** structure is used in conjunction with the [QueryBusInformation](#) interface routine to report information about the bus.

## Syntax

C++

```
typedef struct _USB_BUS_INFORMATION_LEVEL_1 {
    ULONG TotalBandwidth;
    ULONG ConsumedBandwidth;
    ULONG ControllerNameLength;
    WCHAR ControllerNameUnicodeString[1];
} USB_BUS_INFORMATION_LEVEL_1, *PUSB_BUS_INFORMATION_LEVEL_1;
```

## Members

**TotalBandwidth**

Specifies the total bandwidth, in bits per second, available on the bus.

**ConsumedBandwidth**

Specifies the mean bandwidth already in use, in bits per second.

**ControllerNameLength**

Specifies the length of symbolic name for the host controller, in Unicode.

**ControllerNameUnicodeString[1]**

Specifies the symbolic name for the host controller, in Unicode.

## Remarks

Caller must set the *Level* parameter of the [QueryBusInformation](#) routine to 1.

For more information about how to obtain the proper level of USB interface, see [USB Routines](#).

## Requirements

<b>Minimum supported client</b>	Available in Microsoft Windows XP and later operating systems.
<b>Header</b>	usbbusif.h (include Usbbusif.h)

## See also

[QueryBusInformation](#)

[USB Structures](#)

# USB\_BUS\_INTERFACE\_USBDI\_V0 structure (usbbusif.h)

Article04/01/2021

The **USB\_BUS\_INTERFACE\_USBDI\_V0** structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## Syntax

C++

```
typedef struct _USB_BUS_INTERFACE_USBDI_V0 {
    USHORT             Size;
    USHORT             Version;
    PVOID              BusContext;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    PUSB_BUSIFFN_GETUSBDI_VERSION     GetUSBDIVersion;
    PUSB_BUSIFFN_QUERY_BUS_TIME       QueryBusTime;
    PUSB_BUSIFFN_SUBMIT_ISO_OUT_URB   SubmitIsoOutUrb;
    PUSB_BUSIFFN_QUERY_BUS_INFORMATION QueryBusInformation;
} USB_BUS_INTERFACE_USBDI_V0, *PUSB_BUS_INTERFACE_USBDI_V0;
```

## Members

### Size

Specifies the size in bytes of the buffer that holds the interface pointers.

### Version

Indicates, on input, the version of the interface. This member should have one of the following values:

Value	Meaning
USB_BUSIF_USBDI_VERSION_0	Version 0 of the interface.
USB_BUSIF_USBDI_VERSION_1	Version 1 of the interface.
USB_BUSIF_USBDI_VERSION_2	Version 2 of the interface.
USB_BUSIF_USBDI_VERSION_3	Version 3 of the interface.

### **BusContext**

Contains information that describes the USB bus and the USB bus driver that exposes this interface. This is an opaque entity that the caller must pass to the interface routines.

### **InterfaceReference**

Pointer to a routine that increments the number of references to this interface. For more information about this routine, see [InterfaceReference](#).

### **InterfaceDereference**

Pointer to a routine that decrements the number of references to this interface. For more information about this routine, see [InterfaceDereference](#).

### **GetUSBDIVersion**

Pointer to a routine that returns the USB interface version number, the version number of USB specification that defines the interface, along with host controller capabilities information. This routine returns the highest USBDI interface version that is supported by the port driver. For more information about this routine, see [GetUSBDIVersion](#).

### **QueryBusTime**

Pointer to a routine that returns the current 32-bit USB frame number. This routine replaces the **USBD\_QueryBusTime** function provided by usbd.sys. For more information about this routine, see [QueryBusTime](#).

### **SubmitIsoOutUrb**

Reserved. Do not use.

### **QueryBusInformation**

Pointer to a routine that returns information about the bus. The information that is returned depends on the value of the **Level** member. If **Level** is 0, this routine returns bus bandwidth information. If **Level** is 1, it returns bus bandwidth information and the host controller's symbolic name. This routine replaces the **USBD\_QueryBusInformation** function provided by usbd.sys. For more information about this routine, see [QueryBusInformation](#).

## **Remarks**

For information about how to query for these interfaces, see [Querying for USB Interfaces](#). Callers of the routines in this structure can be running at IRQL <= DISPATCH\_LEVEL.

## Requirements

Header	usbbusif.h (include Usbbusif.h)
--------	---------------------------------

## See also

[Bus Driver Interface Routines for USB Client Drivers](#)

[USB Structures](#)

# USB\_BUS\_INTERFACE\_USBDI\_V1 structure (usbbusif.h)

Article04/01/2021

The **USB\_BUS\_INTERFACE\_USBDI\_V1** structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## Syntax

C++

```
typedef struct _USB_BUS_INTERFACE_USBDI_V1 {
    USHORT             Size;
    USHORT             Version;
    PVOID              BusContext;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    PUSB_BUSIFFN_GETUSBDI_VERSION     GetUSBDIVersion;
    PUSB_BUSIFFN_QUERY_BUS_TIME       QueryBusTime;
    PUSB_BUSIFFN_SUBMIT_ISO_OUT_URB   SubmitIsoOutUrb;
    PUSB_BUSIFFN_QUERY_BUS_INFORMATION QueryBusInformation;
    PUSB_BUSIFFN_IS_DEVICE_HIGH_SPEED IsDeviceHighSpeed;
} USB_BUS_INTERFACE_USBDI_V1, *PUSB_BUS_INTERFACE_USBDI_V1;
```

## Members

### Size

Specifies the size in bytes of the buffer that holds the interface pointers.

### Version

Indicates, on input, the version of the interface. The values that this member can take are as follows.

Value	Meaning
USB_BUSIF_USBDI_VERSION_0	Version 0 of the interface.
USB_BUSIF_USBDI_VERSION_1	Version 1 of the interface.
USB_BUSIF_USBDI_VERSION_2	Version 2 of the interface.
USB_BUSIF_USBDI_VERSION_3	Version 3 of the interface.

---

#### `BusContext`

Contains information that describes the USB bus and the USB bus driver that exposes this interface. This is an opaque entity that the caller must pass to the interface routines.

#### `InterfaceReference`

Pointer to a routine that increments the number of references to this interface. For more information about this routine, see [InterfaceReference](#).

#### `InterfaceDereference`

Pointer to a routine that decrements the number of references to this interface. For more information about this routine, see [InterfaceDereference](#).

#### `GetUSBDIVersion`

Pointer to a routine that returns the USB interface version number, the version number of USB specification that defines the interface, along with host controller capabilities information. This routine returns the highest USBDI interface version supported by the port driver. For more information about this routine, see [GetUSBDIVersion](#).

#### `QueryBusTime`

Pointer to a routine that returns the current 32-bit USB frame number. This routine replaces the **USBD\_QueryBusTime** function provided by usbd.sys. For more information about this routine, see [QueryBusTime](#).

#### `SubmitIsoOutUrb`

Reserved. Do not use.

#### `QueryBusInformation`

Pointer to a routine that retrieves information about the bus. The information that is returned depends on the value of the **Level** member. If **Level** is 0, this routine returns bus bandwidth information. If **Level** is 1, it returns bus bandwidth information and the host controller's symbolic name. This routine replaces the **USBD\_QueryBusInformation** function provided by usbd.sys. For more information about this routine, see [QueryBusInformation](#).

#### `IsDeviceHighSpeed`

Pointer to a routine that determines whether the USB device is operating at high speed. This routine returns **TRUE** if the USB device is operating at high speed (USB 2.0-

compliant device). Otherwise this routine returns FALSE. For more information about this routine, see [IsDeviceHighSpeed](#).

## Remarks

The **IsDeviceHighSpeed** member does not indicate whether a device is capable of high speed operation, but rather whether it is in fact operating at high speed.

The routines in this structure must be callable at IRQL >= DISPATCH\_LEVEL.

## Requirements

Minimum supported client	Available in Microsoft Windows XP and later operating systems.
Header	usbbusif.h (include Usbbusif.h)

## See also

[Bus Driver Interface Routines for USB Client Drivers](#)

[USB Structures](#)

# USB\_BUS\_INTERFACE\_USBDI\_V2 structure (usbbusif.h)

Article04/01/2021

The **USB\_BUS\_INTERFACE\_USBDI\_V2** structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## Syntax

C++

```
typedef struct _USB_BUS_INTERFACE_USBDI_V2 {
    USHORT             Size;
    USHORT             Version;
    PVOID              BusContext;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    PUSB_BUSIFFN_GETUSBDI_VERSION     GetUSBDIVersion;
    PUSB_BUSIFFN_QUERY_BUS_TIME       QueryBusTime;
    PUSB_BUSIFFN_SUBMIT_ISO_OUT_URB   SubmitIsoOutUrb;
    PUSB_BUSIFFN_QUERY_BUS_INFORMATION QueryBusInformation;
    PUSB_BUSIFFN_IS_DEVICE_HIGH_SPEED IsDeviceHighSpeed;
    PUSB_BUSIFFN_ENUM_LOG_ENTRY       EnumLogEntry;
} USB_BUS_INTERFACE_USBDI_V2, *PUSB_BUS_INTERFACE_USBDI_V2;
```

## Members

Size

Specifies the size in bytes of the buffer that holds the interface pointers.

Version

Indicates, on input, the version of the interface. The values that this member can take are as follows.

Value	Meaning
USB_BUSIF_USBDI_VERSION_0	Version 0 of the interface.
USB_BUSIF_USBDI_VERSION_1	Version 1 of the interface.
USB_BUSIF_USBDI_VERSION_2	Version 2 of the interface.

**BusContext**

Contains information that describes the USB bus and the USB bus driver that exposes this interface. This is an opaque entity that the caller must pass to the interface routines.

**InterfaceReference**

Pointer to a routine that increments the number of references to this interface. For more information about this routine, see [InterfaceReference](#).

**InterfaceDereference**

Pointer to a routine that decrements the number of references to this interface. For more information about this routine, see [InterfaceDereference](#).

**GetUSBDIVersion**

Pointer to a routine that returns the USB interface version number, the version number of USB specification that defines the interface, along with host controller capabilities information. This routine returns the highest USBDI interface version supported by the port driver. For more information about this routine, see [GetUSBDIVersion](#).

**QueryBusTime**

Pointer to a routine that returns the current 32-bit USB frame number. This routine replaces the **USBD\_QueryBusTime** function provided by usbd.sys. For more information about this routine, see [QueryBusTime](#).

**SubmitIsoOutUrb**

Reserved. Do not use.

**QueryBusInformation**

Pointer to a routine that retrieves information about the bus. The information that is returned depends on the value of the **Level** member. If **Level** is 0, this routine returns bus bandwidth information. If **Level** is 1, it returns bus bandwidth information and the host controller's symbolic name. This routine replaces the **USBD\_QueryBusInformation** function provided by usbd.sys. For more information about this routine, see [QueryBusInformation](#).

**IsDeviceHighSpeed**

Pointer to a routine that determines if the USB device is operating at high speed. This routine returns **TRUE** if the USB device is operating at high speed USB 2.0 compliant device. Returns **FALSE** otherwise. For more information about this routine, see [IsDeviceHighSpeed](#).

#### EnumLogEntry

Reserved. Do not use.

## Remarks

The **IsDeviceHighSpeed** routine does not indicate whether a device is capable of high-speed operation, but whether it is in fact operating at high speed.

The routines in this structure must be callable at IRQL >= DISPATCH\_LEVEL.

## Requirements

Minimum supported client	Available in Microsoft Windows XP and later operating systems.
Header	usbbusif.h (include Usbbusif.h)

## See also

[Bus Driver Interface Routines for USB Client Drivers](#)

[USB Structures](#)

# USB\_BUS\_INTERFACE\_USBDI\_V3 structure (usbbusif.h)

Article04/01/2021

The **USB\_BUS\_INTERFACE\_USBDI\_V3** structure is provided by the USB hub driver to allow USB clients to make direct calls to the hub driver without allocating IRPs.

## Syntax

C++

```
typedef struct _USB_BUS_INTERFACE_USBDI_V3 {
    USHORT             Size;
    USHORT             Version;
    PVOID              BusContext;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    PUSB_BUSIFFN_GETUSBDI_VERSION     GetUSBDIVersion;
    PUSB_BUSIFFN_QUERY_BUS_TIME       QueryBusTime;
    PUSB_BUSIFFN_SUBMIT_ISO_OUT_URB   SubmitIsoOutUrb;
    PUSB_BUSIFFN_QUERY_BUS_INFORMATION QueryBusInformation;
    PUSB_BUSIFFN_IS_DEVICE_HIGH_SPEED IsDeviceHighSpeed;
    PUSB_BUSIFFN_ENUM_LOG_ENTRY       EnumLogEntry;
    PUSB_BUSIFFN_QUERY_BUS_TIME_EX    QueryBusTimeEx;
    PUSB_BUSIFFN_QUERY_CONTROLLER_TYPE QueryControllerType;
} USB_BUS_INTERFACE_USBDI_V3, *PUSB_BUS_INTERFACE_USBDI_V3;
```

## Members

### Size

Specifies the size in bytes of the buffer that holds the interface pointers.

### Version

Indicates, on input, the version of the interface. The values that this member can take are as follows.

Value	Meaning
USB_BUSIF_USBDI_VERSION_0	Version 0 of the interface.
USB_BUSIF_USBDI_VERSION_1	Version 1 of the interface.

USB_BUSIF_USBDI_VERSION_2	Version 2 of the interface.
USB_BUSIF_USBDI_VERSION_3	Version 3 of the interface.

#### BusContext

Contains information that describes the USB bus and the USB bus driver that exposes this interface. This is an opaque entity that the caller must pass to the interface routines.

#### InterfaceReference

Pointer to a routine that increments the number of references to this interface. For more information about this routine, see [InterfaceReference](#).

#### InterfaceDereference

Pointer to a routine that decrements the number of references to this interface. For more information about this routine, see [InterfaceDereference](#).

#### GetUSBDIVersion

Pointer to a routine that returns the USB interface version number, the version number of USB specification that defines the interface, along with host controller capabilities information. This routine returns the highest USBDI interface version supported by the port driver. For more information about this routine, see [GetUSBDIVersion](#).

#### QueryBusTime

Pointer to a routine that returns the current 32-bit USB frame number. This routine replaces the [USBD\\_QueryBusTime](#) function provided by usbd.sys. For more information about this routine, see [QueryBusTime](#).

#### SubmitIsoOutUrb

Reserved. Do not use.

#### QueryBusInformation

Pointer to a routine that retrieves information about the bus. The information that is returned depends on the value of the **Level** member. If **Level** is 0, this routine returns bus bandwidth information. If **Level** is 1, it returns bus bandwidth information and the host controller's symbolic name. This routine replaces the [USBD\\_QueryBusInformation](#) function provided by usbd.sys. For more information about this routine, see [QueryBusInformation](#).

#### IsDeviceHighSpeed

Pointer to a routine that determines if the USB device is operating at high speed. This routine returns **TRUE** if the USB device is operating at high speed USB 2.0 compliant device. Returns **FALSE** otherwise. For more information about this routine, see [IsDeviceHighSpeed](#).

#### EnumLogEntry

Reserved. Do not use.

#### QueryBusTimeEx

This routine is not implemented.

#### QueryControllerType

Pointer to a routine that returns information about the USB host controller the USB device is attached to. For more information about this routine, see [QueryControllerType](#).

## Remarks

The **IsDeviceHighSpeed** routine does not indicate whether a device is capable of high-speed operation, but whether it is in fact operating at high speed.

The routines in this structure must be callable at IRQL >= DISPATCH\_LEVEL.

## Requirements

Minimum supported client	Windows Vista and later operating systems.
Header	usbbusif.h (include Usbbusif.h)

## See also

[Bus Driver Interface Routines for USB Client Drivers](#)

[USB Structures](#)

# USBC\_DEVICE\_CONFIGURATION\_INTERFACE\_V1 structure (usbbusif.h)

Article02/22/2024

The **USBC\_DEVICE\_CONFIGURATION\_INTERFACE\_V1** structure is exposed by the vendor-supplied filter drivers to assist the USB generic parent driver in defining interface collections.

## Syntax

C++

```
typedef struct _USBC_DEVICE_CONFIGURATION_INTERFACE_V1 {
    USHORT             Size;
    USHORT             Version;
    PVOID              Context;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    USBC_START_DEVICE_CALLBACK StartDeviceCallback;
    USBC_PDO_ENABLE_CALLBACK PdoEnableCallback;
    PVOID              Reserved[7];
} USBC_DEVICE_CONFIGURATION_INTERFACE_V1,
*PUSBC_DEVICE_CONFIGURATION_INTERFACE_V1;
```

## Members

Size

The size, in bytes, of this structure.

Version

The version of the interface.

Context

The USB generic parent driver does not use this member. It is populated by the vendor supplied filter driver and may be used to track instance information for the bus interface. It is passed as a parameter to [InterfaceReference](#) and [InterfaceDereference](#).

InterfaceReference

Pointer to a routine that increments the number of references to this interface. For more information about this routine, see [InterfaceReference](#).

`InterfaceDereference`

Pointer to a routine that decrements the number of references to this interface. For more information about this routine, see [InterfaceDereference](#).

`StartDeviceCallback`

Pointer to the callback routine that the filter driver furnishes to the USB generic parent driver to assist in defining interface collections on a device. For more information, see [USBC\\_START\\_DEVICE\\_CALLBACK](#).

`PdoEnableCallback`

Reserved.

`Reserved[7]`

Reserved.

## Requirements

[Expand table](#)

Requirement	Value
Header	usbbusif.h (include Usbbusif.h)

## See also

[Customizing Enumeration of Interface Collections for Composite Devices](#)

[USB Structures](#)

[USBC\\_START\\_DEVICE\\_CALLBACK](#)

# USBC\_FUNCTION\_DESCRIPTOR structure (usbbusif.h)

Article 02/22/2024

The **USBC\_FUNCTION\_DESCRIPTOR** structure describes a USB function and its associated interface collection.

## Syntax

C++

```
typedef struct _USBC_FUNCTION_DESCRIPTOR {
    UCHAR             FunctionNumber;
    UCHAR             NumberOfInterfaces;
    PUSB_INTERFACE_DESCRIPTOR *InterfaceDescriptorList;
    UNICODE_STRING    HardwareId;
    UNICODE_STRING    CompatiblId;
    UNICODE_STRING    FunctionDescription;
    ULONG             FunctionFlags;
    PVOID             Reserved;
} USBC_FUNCTION_DESCRIPTOR, *PUSBC_FUNCTION_DESCRIPTOR;
```

## Members

**FunctionNumber**

The zero-based index of the interface collection.

**NumberOfInterfaces**

The number of interfaces in the interface collection.

**InterfaceDescriptorList**

An array of pointers to [USB\\_INTERFACE\\_DESCRIPTOR](#)-type structures that describe the interfaces in the interface collection.

**HardwareId**

The hardware identifier of the interface collection.

**CompatiblId**

The compatible identifier of the interface collection.

**FunctionDescription**

A description of the interface collection in human-readable text.

**FunctionFlags**

Vendor-defined flags that describe the interface collection.

**Reserved**

Reserved.

## Remarks

For information on how to use user-defined callback routines to provide a custom definition of the interface collections on a device, see [Customizing Enumeration of Interface Collections for Composite Devices](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbbusif.h (include Usbbusif.h)

## See also

[USB Structures](#)

[USB\\_INTERFACE\\_DESCRIPTOR](#)

# USBC\_START\_DEVICE\_CALLBACK callback function (usbbusif.h)

Article02/22/2024

The **USBC\_START\_DEVICE\_CALLBACK** routine allows a USB client driver to provide a custom definition of the interface collections on a device.

## Syntax

C++

```
USBC_START_DEVICE_CALLBACK UsbcStartDeviceCallback;

NTSTATUS UsbcStartDeviceCallback(
    [in] PUSB_DEVICE_DESCRIPTOR DeviceDescriptor,
    [in] PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    [out] PUSBC_FUNCTION_DESCRIPTOR *FunctionDescriptorBuffer,
    [out] PULONG FunctionDescriptorBufferLength,
    [in] PDEVICE_OBJECT FdoDeviceObject,
    [in] PDEVICE_OBJECT PdoDeviceObject
)
{...}
```

## Parameters

[in] `DeviceDescriptor`

The device descriptor of the device.

[in] `ConfigurationDescriptor`

The configuration of the device.

[out] `FunctionDescriptorBuffer`

Pointer to a buffer that contains an array of function descriptors ([USBC\\_FUNCTION\\_DESCRIPTOR](#)).

[out] `FunctionDescriptorBufferLength`

The length in bytes of the buffer that *FunctionDescriptorBuffer* points to.

[in] `FdoDeviceObject`

The function device object for the device.

[in] PdoDeviceObject

The physical device object for the device.

## Return value

If the operation succeeds, the vendor-supplied callback routine must return STATUS\_SUCCESS.

## Remarks

For a general description of the callback routine mechanism, see [Customizing Enumeration of Interface Collections for Composite Devices](#).

## Requirements

[+] Expand table

Requirement	Value
Target Platform	Desktop
Header	usbbusif.h (include Usbbusif.h)

## See also

[USBC\\_DEVICE\\_CONFIGURATION\\_INTERFACE\\_V1](#)

# usbctypes.h header

Article01/23/2023

This header is the structure and enumeration declarations for client drivers of the USB Policy Manager to monitor the activities of USB Type-C connectors and/or get involved into policy decisions of USB Type-C connectors.

Do not include this header directly. Instead, only include [Usbpmai.h](#).

For more information, see:

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

usbctypes.h contains the following programming interfaces:

## Functions

### [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_GET\\_TYPE](#)

Retrieves the type of Power Data Object (PDO).

## Structures

### [USBC\\_PD\\_ALTERNATE\\_MODE](#)

Stores information about the alternate mode that was detected.

### [USBC\\_PD\\_POWER\\_DATA\\_OBJECT](#)

Describes a power data object (PDO).

### [USBC\\_PD\\_REQUEST\\_DATA\\_OBJECT](#)

Describes a request data object (RDO).

## Enumerations

## [USBC\\_CHARGING\\_STATE](#)

Learn how USBC\_CHARGING\_STATE defines the charging state of a Type-C connector.

## [USBC\\_CURRENT](#)

Learn how USBC\_CURRENT defines different Type-C current levels, as defined in the Type-C specification.

## [USBC\\_DATA\\_ROLE](#)

Defines data roles of USB Type-C connected devices.

## [USBC\\_PARTNER](#)

Defines values for the type of connector partner detected on the USB Type-C connector.

## [USBC\\_PD\\_AUGMENTED\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

Learn how USBC\_PD\_AUGMENTED\_POWER\_DATA\_OBJECT\_TYPE defines augmented power data object (APDO) types.

## [USBC\\_PD\\_CONN\\_STATE](#)

Learn how USBC\_PD\_CONN\_STATE defines power delivery (PD) negotiation states of a Type-C port.

## [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

Learn how USBC\_PD\_POWER\_DATA\_OBJECT\_TYPE defines power data object (PDO) types.

## [USBC\\_POWER\\_ROLE](#)

Learn how USBC\_POWER\_ROLE defines power roles of USB Type-C connected devices.

## [USBC\\_TYPEC\\_OPERATING\\_MODE](#)

Learn how USBC\_TYPEC\_OPERATING\_MODE defines operating modes of a USB Type-C connector.

## [USBC\\_UCSI\\_SET\\_POWER\\_LEVEL\\_C\\_CURRENT](#)

Defines values for current power operation mode.

# USBC\_CHARGING\_STATE enumeration (usbctypes.h)

Article02/22/2024

Defines the charging state of a Type-C connector.

## Syntax

C++

```
typedef enum _USBC_CHARGING_STATE {  
    UsbCChargingStateInvalid,  
    UsbCChargingStateNotCharging,  
    UsbCChargingStateNominalCharging,  
    UsbCChargingStateSlowCharging,  
    UsbCChargingStateTrickleCharging  
} USBC_CHARGING_STATE;
```

## Constants

[+] Expand table

`UsbCChargingStateInvalid`

Indicates the charging state is invalid.

`UsbCChargingStateNotCharging`

Indicates the port is not drawing a charge.

`UsbCChargingStateNominalCharging`

Indicates the port is drawing a nominal charge.

`UsbCChargingStateSlowCharging`

Indicates the port is drawing a slow charge.

`UsbCChargingStateTrickleCharging`

Indicates the port is drawing a trickle charge.

## Requirements

[ ] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_CURRENT enumeration (usbctypes.h)

Article 02/22/2024

Defines different Type-C current levels, as defined in the Type-C specification.

## Syntax

C++

```
typedef enum _USBC_CURRENT {
    UsbCCurrentInvalid,
    UsbCCurrentDefaultUsb,
    UsbCCurrent1500mA,
    UsbCCurrent3000mA
} USBC_CURRENT;
```

## Constants

[] Expand table

<b>UsbCCurrentInvalid</b> Indicates the power sourcing current state is invalid.
<b>UsbCCurrentDefaultUsb</b> Indicates the power sourcing current is the default USB current.
<b>UsbCCurrent1500mA</b> Indicates the power sourcing current is 1500 mA.
<b>UsbCCurrent3000mA</b> Indicates the power sourcing current is 3000 mA.

## Requirements

[] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_DATA\_ROLE enumeration (usbctypes.h)

Article 02/22/2024

Defines data roles of USB Type-C connected devices.

## Syntax

C++

```
typedef enum _USBC_DATA_ROLE {  
    UsbCDataRoleInvalid,  
    UsbCDataRoleDfp,  
    UsbCDataRoleUfp  
} USBC_DATA_ROLE;
```

## Constants

[+] Expand table

<code>UsbCDataRoleInvalid</code>	Indicates the data role state is invalid.
<code>UsbCDataRoleDfp</code>	Operates as a DFP.
<code>UsbCDataRoleUfp</code>	Operates as a UFP.

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	usbctypes.h (include usbctypes.h)

# USBC\_PARTNER enumeration (usbctypes.h)

Article 02/22/2024

Defines values for the type of connector partner detected on the USB Type-C connector.

## Syntax

C++

```
typedef enum _USBC_PARTNER {
    UsbCPartnerInvalid,
    UsbCPartnerUfp,
    UsbCPartnerDfp,
    UsbCPartnerPoweredCableNoUfp,
    UsbCPartnerPoweredCableWithUfp,
    UsbCPartnerAudioAccessory,
    UsbCPartnerDebugAccessory
} USBC_PARTNER;
```

## Constants

[+] Expand table

<b>UsbCPartnerInvalid</b> Invalid partner.
<b>UsbCPartnerUfp</b> The partner is a UFP.
<b>UsbCPartnerDfp</b> The partner is a DFP.
<b>UsbCPartnerPoweredCableNoUfp</b> The partner is not a UFP when attached to powered cable.
<b>UsbCPartnerPoweredCableWithUfp</b> The partner is a UFP when attached to powered cable.
<b>UsbCPartnerAudioAccessory</b> The partner is an audio adapter accessory.

#### UsbCPartnerDebugAccessory

The partner is a debug adapter accessory.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_PD\_ALTERNATE\_MODE structure (usbctypes.h)

Article02/22/2024

Stores information about the alternate mode that was detected.

## Syntax

C++

```
typedef struct _USBC_PD_ALTERNATE_MODE {
    UINT16 SVID;
    UINT32 Mode;
} USBC_PD_ALTERNATE_MODE, *PUSBC_PD_ALTERNATE_MODE;
```

## Members

SVID

The Standard or Vendor ID (SVID) for the alternate mode that was entered.

Mode

The Standard or Vendor defined Mode value for the alternate mode that was entered.

## Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_PD\_AUGMENTED\_POWER\_DATA\_OBJECT\_TYPE enumeration (usbctypes.h)

Article02/22/2024

Defines augmented power data object (APDO) types.

## Syntax

C++

```
typedef enum _USBC_PD_AUGMENTED_POWER_DATA_OBJECT_TYPE {
    UsbCPdApdoTypeProgrammablePowerSupply
} USBC_PD_AUGMENTED_POWER_DATA_OBJECT_TYPE;
```

## Constants

[] Expand table

`UsbCPdApdoTypeProgrammablePowerSupply`

Indicates the PD power data object is a programmable power supply augmented power data object (APDO)

## Remarks

For information, see the [Power Delivery specification](#).

## Requirements

[] Expand table

Requirement	Value
Header	usbctypes.h

## See also

- [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

# USBC\_PD\_CONN\_STATE enumeration (usbctypes.h)

Article 02/22/2024

Defines power delivery (PD) negotiation states of a Type-C port.

## Syntax

C++

```
typedef enum _USBC_PD_CONN_STATE {  
    UsbCPdConnStateInvalid,  
    UsbCPdConnStateNotSupported,  
    UsbCPdConnStateNegotiationFailed,  
    UsbCPdConnStateNegotiationSucceeded  
} USBC_PD_CONN_STATE;
```

## Constants

[] Expand table

<code>UsbCPdConnStateInvalid</code> Indicates the PD negotiation state is invalid.
<code>UsbCPdConnStateNotSupported</code> Indicates a PD connection is not supported.
<code>UsbCPdConnStateNegotiationFailed</code> Indicates the PD negotiation failed.
<code>UsbCPdConnStateNegotiationSucceeded</code> Indicates the PD negotiation succeeded.

## Requirements

[] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_PD\_POWER\_DATA\_OBJECT union (usbctypes.h)

Article02/09/2022

Describes a power data object (PDO). For information about these members, see the [Power Delivery specification](#).

## Syntax

C++

```
typedef union _USBC_PD_POWER_DATA_OBJECT {
    UINT32 U;
    struct {
        UINT32 Reserved : 30;
        UINT32 Type : 2;
    } Common;
    struct {
        UINT32 MaximumCurrentIn10mA : 10;
        UINT32 VoltageIn50mV : 10;
        UINT32 PeakCurrent : 2;
        UINT32 Reserved1 : 3;
        UINT32 DataRoleSwap : 1;
        UINT32 UsbCommunicationCapable : 1;
        UINT32 ExternallyPowered : 1;
        UINT32 UsbSuspendSupported : 1;
        UINT32 DualRolePower : 1;
        UINT32 FixedSupply : 2;
    } FixedSupplyPdo;
    struct {
        UINT32 MaximumAllowablePowerIn250mW : 10;
        UINT32 MinimumVoltageIn50mV : 10;
        UINT32 MaximumVoltageIn50mV : 10;
        UINT32 Battery : 2;
    } BatterySupplyPdo;
    struct {
        UINT32 MaximumCurrentIn10mA : 10;
        UINT32 MinimumVoltageIn50mV : 10;
        UINT32 MaximumVoltageIn50mV : 10;
        UINT32 VariableSupportNonBattery : 2;
    } VariableSupplyNonBatteryPdo;
    struct {
        UINT32 MaximumCurrentIn50mA : 7;
        UINT32 Reserved1 : 1;
        UINT32 MinimumVoltageIn100mV : 8;
        UINT32 Reserved2 : 1;
        UINT32 MaximumVoltageIn100mV : 8;
        UINT32 Reserved3 : 2;
    } VariableSupplyNonBatteryPdo;
}
```

```

    UINT32 PpsPowerLimited : 1;
    UINT32 AugmentedPowerDataObjectType : 2;
    UINT32 AugmentedPowerDataObject : 2;
} ProgrammablePowerSupplyApdo;
struct {
    UINT32 OperationalCurrentIn10mA : 10;
    UINT32 VoltageIn50mV : 10;
    UINT32 Reserved : 5;
    UINT32 DataRoleSwap : 1;
    UINT32 UsbCommunicationCapable : 1;
    UINT32 ExternallyPowered : 1;
    UINT32 HigherCapability : 1;
    UINT32 DualRolePower : 1;
    UINT32 FixedSupply : 2;
} FixedSupplyPdoSink;
struct {
    UINT32 OperationalPowerIn250mW : 10;
    UINT32 MinimumVoltageIn50mV : 10;
    UINT32 MaximumVoltageIn50mV : 10;
    UINT32 Battery : 2;
} BatterySupplyPdoSink;
struct {
    UINT32 OperationalCurrentIn10mA : 10;
    UINT32 MinimumVoltageIn50mV : 10;
    UINT32 MaximumVoltageIn50mV : 10;
    UINT32 VariableSupportNonBattery : 2;
} VariableSupplyNonBatteryPdoSink;
} USBC_PD_POWER_DATA_OBJECT, *PUSBC_PD_POWER_DATA_OBJECT;

```

## Members

U

Size of the structure.

Common

Common.Reserved

Reserved.

Common.Type

Type of power data object.

FixedSupplyPdo

Describing a fixed supply type power data object.

FixedSupplyPdo.MaximumCurrentIn10mA

Maximum current in multiples of 10 mA.

`FixedSupplyPdo.VoltageIn50mV`

Voltage in multiples of 50 mV.

`FixedSupplyPdo.Pea`

Peak current.

`FixedSupplyPdo.Reserved1`

Reserved for future use.

`FixedSupplyPdo.DataRoleSwap`

If set, indicates the power data object can perform a data role swap.

`FixedSupplyPdo.UsbCommunicationCapable`

If set, indicates the power data object is USB communication capable.

`FixedSupplyPdo.ExternallyPowered`

If set, indicates the power data object is externally powered.

`FixedSupplyPdo.UsbSuspendSupported`

Indicates support for USB suspend.

`FixedSupplyPdo.DualRolePower`

Dual role power.

`FixedSupplyPdo.FixedSupply`

Fixed supply.

`BatterySupplyPdo`

Contains bitfields describing a variable-supply non-battery PD object.

`BatterySupplyPdo.MaximumAllowablePowerIn250mW`

Describes the maximum voltage in multiples of 250mV.

`BatterySupplyPdo.MinimumVoltageIn50mV`

Describes the minimum voltage in multiples of 50mV.

`BatterySupplyPdo.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 50mV.

`BatterySupplyPdo.Battery`

Battery type.

`VariableSupplyNonBatteryPdo`

Contains bitfields describing a variable-supply non-battery PD object.

`VariableSupplyNonBatteryPdo.MaximumCurrentIn10mA`

Describes the maximum current in multiples of 10 mA.

`VariableSupplyNonBatteryPdo.MinimumVoltageIn50mV`

Describes the minimum current in multiples of 50 mA.

`VariableSupplyNonBatteryPdo.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 10 mA.

`VariableSupplyNonBatteryPdo.VariableSupportNonBattery`

Variable Support Non Battery type.

`ProgrammablePowerSupplyApdo`

Describing a programmable power supply augmented power delivery object.

`ProgrammablePowerSupplyApdo.MaximumCurrentIn50mA`

Describes the maximum current in multiples of 50 mA.

`ProgrammablePowerSupplyApdo.Reserved1`

Reserved, do not use.

`ProgrammablePowerSupplyApdo.MinimumVoltageIn100mV`

Describes the minimum voltage in multiples of 100 mV.

`ProgrammablePowerSupplyApdo.Reserved2`

Reserved, do not use.

`ProgrammablePowerSupplyApdo.MaximumVoltageIn100mV`

Describes the maximum voltage in multiples of 100 mV.

`ProgrammablePowerSupplyApdo.Reserved3`

Reserved, do not use.

`ProgrammablePowerSupplyApdo.PpsPowerLimited`

Power limited supply.

`ProgrammablePowerSupplyApdo.AugmentedPowerDataObjectType`

Describes a USBC augmented power data object type.

`ProgrammablePowerSupplyApdo.AugmentedPowerDataObject`

Describes a USBC power data object type.

`FixedSupplyPdoSink`

Describing a fixed supply type power data object.

`FixedSupplyPdoSink.OperationalCurrentIn10mA`

Describes the operational current in multiples of 10 mA.

`FixedSupplyPdoSink.VoltageIn50mV`

Voltage in multiples of 50 mV.

`FixedSupplyPdoSink.Reserved`

Reserved.

`FixedSupplyPdoSink.DataRoleSwap`

If set, indicates the power data object can perform a data role swap.

`FixedSupplyPdoSink.UsbCommunicationCapable`

If set, indicates the power data object is USB communication capable.

`FixedSupplyPdoSink.ExternallyPowered`

If set, indicates the power data object is externally powered.

`FixedSupplyPdoSink.HigherCapability`

Power data object has a higher capability.

`FixedSupplyPdoSink.DualRolePower`

Dual role power.

`FixedSupplyPdoSink.FixedSupply`

Fixed supply.

`BatterySupplyPdoSink`

Contains bitfields describing a variable-supply non-battery PD object.

`BatterySupplyPdoSink.OperationalPowerIn250mW`

Describes the maximum voltage in multiples of 250mV.

`BatterySupplyPdoSink.MinimumVoltageIn50mV`

Describes the minimum voltage in multiples of 50mV.

`BatterySupplyPdoSink.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 50mV.

`BatterySupplyPdoSink.Battery`

Battery type.

`VariableSupplyNonBatteryPdoSink`

Contains bit fields describing a battery supply power data object.

`VariableSupplyNonBatteryPdoSink.OperationalCurrentIn10mA`

Describes the operational current in multiples of 10 mA.

`VariableSupplyNonBatteryPdoSink.MinimumVoltageIn50mV`

Describes the minimum voltage in multiples of 50mV.

`VariableSupplyNonBatteryPdoSink.MaximumVoltageIn50mV`

Describes the maximum voltage in multiples of 50mV.

## VariableSupplyNonBatteryPdoSink.VariableSupportNonBattery

Variable Support Non Battery type.

# Requirements

Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_PD\_POWER\_DATA\_OBJECT\_GET\_TYPE function (usbctypes.h)

Article02/22/2024

Retrieves the type of Power Data Object (PDO).

## Syntax

C++

```
USBC_PD_POWER_DATA_OBJECT_TYPE USBC_PD_POWER_DATA_OBJECT_GET_TYPE(  
    PUSBC_PD_POWER_DATA_OBJECT Pdo  
) ;
```

## Parameters

Pdo

A pointer to a [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#) structure that contains the type of PDO.

## Return value

This function returns the type in `Common.Type` member of [USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#).

## Requirements

 Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

## See also

[USBC\\_PD\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

# USBC\_PD\_POWER\_DATA\_OBJECT\_TYPE enumeration (usbctypes.h)

Article 02/22/2024

Defines power data object (PDO) types.

## Syntax

C++

```
typedef enum _USBC_PD_POWER_DATA_OBJECT_TYPE {
    UsbCPdPdoTypeFixedSupply,
    UsbCPdPdoTypeBatterySupply,
    UsbCPdPdoTypeVariableSupplyNonBattery,
    UsbCPdPdoTypeAugmentedPowerDataObject
} USBC_PD_POWER_DATA_OBJECT_TYPE;
```

## Constants

[ ] [Expand table](#)

<code>UsbCPdPdoTypeFixedSupply</code> Indicates the PD data object type is a fixed supply.
<code>UsbCPdPdoTypeBatterySupply</code> Indicates the PD data object type is a battery supply.
<code>UsbCPdPdoTypeVariableSupplyNonBattery</code> Indicates the PD data object type is a non-battery variable supply.
<code>UsbCPdPdoTypeAugmentedPowerDataObject</code> Indicates the PD data object type is an augmented supply.

## Remarks

For information about these members, see the [Power Delivery specification ↗](#).

## Requirements

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

## See also

- [\\_USBC\\_PD\\_AUGMENTED\\_POWER\\_DATA\\_OBJECT\\_TYPE](#)

# USBC\_PD\_REQUEST\_DATA\_OBJECT union (usbctypes.h)

Article02/09/2022

Describes a request data object (RDO). For information about these members, see the [Power Delivery specification](#).

## Syntax

C++

```
typedef union _USBC_PD_REQUEST_DATA_OBJECT {
    UINT32 U;
    struct {
        UINT32 Reserved1 : 28;
        UINT32 ObjectPosition : 3;
        UINT32 Reserved2 : 1;
    } Common;
    struct {
        UINT32 MaximumOperatingCurrentIn10mA : 10;
        UINT32 OperatingCurrentIn10mA : 10;
        UINT32 Reserved1 : 6;
        UINT32 CapabilityMismatch : 1;
        UINT32 GiveBackFlag : 1;
        UINT32 ObjectPosition : 3;
        UINT32 Reserved2 : 1;
    } FixedAndVariableRdo;
    struct {
        UINT32 MaximumOperatingPowerIn250mW : 10;
        UINT32 OperatingPowerIn250mW : 10;
        UINT32 Reserved1 : 6;
        UINT32 CapabilityMismatch : 1;
        UINT32 GiveBackFlag : 1;
        UINT32 ObjectPosition : 3;
        UINT32 Reserved2 : 1;
    } BatteryRdo;
    struct {
        UINT32 OperatingCurrentIn50mA : 7;
        UINT32 Reserved1 : 2;
        UINT32 OutputVoltageIn20mV : 11;
        UINT32 Reserved2 : 3;
        UINT32 UnchunkedExtendedMessagesSupported : 1;
        UINT32 Reserved3 : 2;
        UINT32 CapabilityMismatch : 1;
        UINT32 Reserved4 : 1;
        UINT32 ObjectPosition : 3;
        UINT32 Reserved5 : 1;
    } OtherRdo;
}
```

```
    } ProgrammableRdo;  
} USBC_PD_REQUEST_DATA_OBJECT, *PUSBC_PD_REQUEST_DATA_OBJECT;
```

## Members

U

Size of the structure.

Common

Common.Reserved1

Reserved, do not use.

Common.ObjectPosition

Object position.

Common.Reserved2

Reserved, do not use.

FixedAndVariableRdo

Contains bit fields describing a request data object.

FixedAndVariableRdo.MaximumOperatingCurrentIn10mA

Maximum current in 10 mA units.

FixedAndVariableRdo.OperatingCurrentIn10mA

Operating current in 10 mA units.

FixedAndVariableRdo.Reserved1

Reserved, do not use.

FixedAndVariableRdo.CapabilityMismatch

Capability mismatch

FixedAndVariableRdo.GiveBackFlag

Giveback flag.

`FixedAndVariableRdo.ObjectPosition`

Object position.

`FixedAndVariableRdo.Reserved2`

Reserved, do not use.

`BatteryRdo`

Contains bit fields describing a request data object.

`BatteryRdo.MaximumOperatingPowerIn250mW`

Maximum operating power in 250 mW units.

`BatteryRdo.OperatingPowerIn250mW`

Operating power in 250 mW units.

`BatteryRdo.Reserved1`

Reserved, do not use.

`BatteryRdo.CapabilityMismatch`

Capability mismatch.

`BatteryRdo.GiveBackFlag`

Giveback flag.

`BatteryRdo.ObjectPosition`

Object position.

`BatteryRdo.Reserved2`

Reserved, do not use.

`ProgrammableRdo`

Describes a programmable request data object.

`ProgrammableRdo.OperatingCurrentIn50mA`

Operating current in 50 mA units.

`ProgrammableRdo.Reserved1`

Reserved, do not use.

`ProgrammableRdo.OutputVoltageIn20mV`

Output voltage in 20 mV units.

`ProgrammableRdo.Reserved2`

Reserved, do not use.

`ProgrammableRdo.UnchunkedExtendedMessagesSupported`

Supports unchunked extended messages.

`ProgrammableRdo.Reserved3`

Reserved, do not use.

`ProgrammableRdo.CapabilityMismatch`

Capability mismatch.

`ProgrammableRdo.Reserved4`

Reserved, do not use.

`ProgrammableRdo.ObjectPosition`

Object position.

`ProgrammableRdo.Reserved5`

Reserved, do not use.

## Requirements

<b>Minimum KMDF version</b>	1.27
<b>Minimum UMDF version</b>	N/A
<b>Header</b>	usbctypes.h (include usbctypes.h)

# USBC\_POWER\_ROLE enumeration (usbctypes.h)

Article02/22/2024

Defines power roles of USB Type-C connected devices.

## Syntax

C++

```
typedef enum _USBC_POWER_ROLE {
    UsbCPowerRoleInvalid,
    UsbCPowerRoleSink,
    UsbCPowerRoleSource
} USBC_POWER_ROLE;
```

## Constants

[+] Expand table

<code>UsbCPowerRoleInvalid</code>	Indicates the power role state is invalid.
<code>UsbCPowerRoleSink</code>	Indicates the power role is set to sink power.
<code>UsbCPowerRoleSource</code>	Indicates the power role is set to source power.

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A

Requirement	Value
Header	usbctypes.h (include usbctypes.h)

# USBC\_TYPEC\_OPERATING\_MODE enumeration (usbctypes.h)

Article 02/22/2024

Defines operating modes of a USB Type-C connector.

## Syntax

C++

```
typedef enum _USBC_TYPEC_OPERATING_MODE {
    UsbCOperatingModeInvalid,
    UsbCOperatingModeDfp,
    UsbCOperatingModeUfp,
    UsbCOperatingModeDrp
} USBC_TYPEC_OPERATING_MODE;
```

## Constants

[] Expand table

<code>UsbCOperatingModeInvalid</code> Indicates the operating mode is invalid.
<code>UsbCOperatingModeDfp</code> Indicates the operating mode is set to downstream-facing port.
<code>UsbCOperatingModeUfp</code> Indicates the operating mode is set to upstream-facing port.
<code>UsbCOperatingModeDrp</code> Indicates the operating mode is set to dual-role port.

## Requirements

[] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# USBC\_UCSI\_SET\_POWER\_LEVEL\_C\_CURR ENT enumeration (usbctypes.h)

Article 02/22/2024

Defines values for current power operation mode.

## Syntax

C++

```
typedef enum _USBC_UCSI_SET_POWER_LEVEL_C_CURRENT {
    UsbCUcsiSetPowerLevelCCurrentPpmDefault,
    UsbCUcsiSetPowerLevelCCurrent3000mA,
    UsbCUcsiSetPowerLevelCCurrent1500mA,
    UsbCUcsiSetPowerLevelCCurrentDefaultUsb
} USBC_UCSI_SET_POWER_LEVEL_C_CURRENT,
*PUSBPM_UCSI_SET_POWER_LEVEL_C_CURRENT;
```

## Constants

[+] Expand table

UsbCUcsiSetPowerLevelCCurrentPpmDefault

USB Type-C default value.

UsbCUcsiSetPowerLevelCCurrent3000mA

USB Type-C current is 3A.

UsbCUcsiSetPowerLevelCCurrent1500mA

USB Type-C current is 3A.

UsbCUcsiSetPowerLevelCCurrentDefaultUsb

Default value for USB operation.

## Requirements

[+] Expand table

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	N/A
Header	usbctypes.h (include usbctypes.h)

# usbdlib.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usbdlib.h contains the following programming interfaces:

## Functions

### [COMPOSITE\\_DEVICE\\_CAPABILITIES\\_INIT](#)

The COMPOSITE\_DEVICE\_CAPABILITIES\_INIT macro initializes the COMPOSITE\_DEVICE\_CAPABILITIES structure.

### [GET\\_ISO\\_URB\\_SIZE](#)

The GET\_ISO\_URB\_SIZE macro returns the number of bytes required to hold an isochronous transfer request.

### [UsbBuildGetStatusRequest](#)

The UsbBuildGetStatusRequest macro formats an URB to obtain status from a device, interface, endpoint, or other device-defined target on a USB device.

### [UsbBuildInterruptOrBulkTransferRequest](#)

The UsbBuildInterruptOrBulkTransferRequest macro formats an URB to send or receive data on a bulk pipe, or to receive data from an interrupt pipe.

### [UsbBuildOpenStaticStreamsRequest](#)

The UsbBuildOpenStaticStreamsRequest inline function formats an URB structure for an open-streams request. The request opens streams associated with the specified bulk endpoint.

### [USBD\\_AssignUrbToIoStackLocation](#)

The USBD\_AssignUrbToIoStackLocation routine is called by a client driver to associate an URB with the IRP's next stack location.

### [USBD\\_BuildRegisterCompositeDevice](#)

The USBD\_BuildRegisterCompositeDevice routine is called by the driver of a USB multi-function

device (composite driver) to initialize a REGISTER\_COMPOSITE\_DEVICE structure with the information required for registering the driver with the USB driver stack.

#### [USBD\\_CalculateUsbBandwidth](#)

The USBD\_CalculateUsbBandwidth routine has been deprecated in Windows XP and later operating systems. Do not use.

#### [USBD\\_CloseHandle](#)

The USBD\_CloseHandle routine is called by a USB client driver to close a USBD handle and release all resources associated with the driver's registration.

#### [USBD\\_CreateConfigurationRequest](#)

The USBD\_CreateConfigurationRequest routine has been deprecated. Use USBD\_CreateConfigurationRequestEx instead.

#### [USBD\\_CreateConfigurationRequestEx](#)

The USBD\_CreateConfigurationRequestEx routine allocates and formats a URB to select a configuration for a USB device.USBD\_CreateConfigurationRequestEx replaces USBD\_CreateConfigurationRequest.

#### [USBD\\_CreateHandle](#)

The USBD\_CreateHandle routine is called by a WDM USB client driver to obtain a USBD handle. The routine registers the client driver with the underlying USB driver stack.

#### [USBD\\_GetInterfaceLength](#)

The USBD\_GetInterfaceLength routine obtains the length of a given interface descriptor, including the length of all endpoint descriptors contained within the interface.

#### [USBD\\_GetPdoRegistryParameter](#)

The USBD\_GetPdoRegistryParameter routine retrieves the value from the specified key in the USB device's hardware registry.

#### [USBD\\_GetUSBDiversion](#)

The USBD\_GetUSBDiversion routine returns version information about the host controller driver (HCD) that controls the client's USB device.Note USBD\_IsInterfaceVersionSupported replaces the USBD\_GetUSBDiversion routine

#### [USBD\\_IsInterfaceVersionSupported](#)

The USBD\_IsInterfaceVersionSupported routine is called by a USB client driver to check whether the underlying USB driver stack supports a particular USBD interface version.

## [USBD\\_IsochUrbAllocate](#)

The USBD\_IsochUrbAllocate routine allocates and formats a URB structure for an isochronous transfer request.

## [USBD\\_ParseConfigurationDescriptor](#)

The USBD\_ParseConfigurationDescriptor routine has been deprecated. Use USBD\_ParseConfigurationDescriptorEx instead.

## [USBD\\_ParseConfigurationDescriptorEx](#)

The USBD\_ParseConfigurationDescriptorEx routine searches a given configuration descriptor and returns a pointer to an interface that matches the given search criteria.

## [USBD\\_ParseDescriptors](#)

The USBD\_ParseDescriptors routine searches a given configuration descriptor and returns a pointer to the first descriptor that matches the search criteria.

## [USBD\\_QueryBusTime](#)

The USBD\_QueryBusTime routine has been deprecated in Windows XP and later operating systems. Do not use.

## [USBD\\_QueryUsbCapability](#)

The USBD\_QueryUsbCapability routine is called by a WDM client driver to determine whether the underlying USB driver stack and the host controller hardware support a specific capability.

## [USBD\\_RegisterHcFilter](#)

The USBD\_RegisterHcFilter routine has been deprecated in Windows XP and later operating systems.

## [USBD\\_SelectConfigUrbAllocateAndBuild](#)

The USBD\_SelectConfigUrbAllocateAndBuild routine allocates and formats a URB structure that is required to select a configuration for a USB device.

## [USBD\\_SelectInterfaceUrbAllocateAndBuild](#)

The USBD\_SelectInterfaceUrbAllocateAndBuild routine allocates and formats a URB structure that is required for a request to select an interface or change its alternate setting.

## [USBD\\_UrbAllocate](#)

The USBD\_UrbAllocate routine allocates a USB Request Block (URB).

### [USBD\\_UrbFree](#)

The USBD\_UrbFree routine releases the URB that is allocated by USBD\_UrbAllocate, USBD\_IsochUrbAllocate, USBD\_SelectConfigUrbAllocateAndBuild, or USBD\_SelectInterfaceUrbAllocateAndBuild.

### [USBD\\_ValidateConfigurationDescriptor](#)

The USBD\_ValidateConfigurationDescriptor routine validates all descriptors returned by a device in its response to a configuration descriptor request.

## Structures

### [COMPOSITE\\_DEVICE\\_CAPABILITIES](#)

The COMPOSITE\_DEVICE\_CAPABILITIES structure specifies the capabilities of the driver of a USB multi-function device (composite driver). To initialize the structure, use the COMPOSITE\_DEVICE\_CAPABILITIES\_INIT macro.

### [REGISTER\\_COMPOSITE\\_DEVICE](#)

The REGISTER\_COMPOSITE\_DEVICE structure is used with the IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE I/O control request to register a parent driver of a Universal Serial Bus (USB) multi-function device (composite driver) with the USB driver stack.

### [REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#)

The purpose of the REQUEST\_REMOTE\_WAKE\_NOTIFICATION structure is to specify input parameters for the IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION I/O control request.

### [USBD\\_INTERFACE\\_LIST\\_ENTRY](#)

The USBD\_INTERFACE\_LIST\_ENTRY structure is used by USB client drivers to create an array of interfaces to be inserted into a configuration request.

# COMPOSITE\_DEVICE\_CAPABILITIES structure (usbdlib.h)

Article02/22/2024

The **COMPOSITE\_DEVICE\_CAPABILITIES** structure specifies the capabilities of the driver of a USB multi-function device (composite driver). To initialize the structure, use the [COMPOSITE\\_DEVICE\\_CAPABILITIES\\_INIT](#) macro.

## Syntax

C++

```
typedef struct _COMPOSITE_DEVICE_CAPABILITIES {
    ULONG CapabilityFunctionSuspend : 1;
    ULONG Reserved : 31;
} COMPOSITE_DEVICE_CAPABILITIES, *PCOMPOSITE_DEVICE_CAPABILITIES;
```

## Members

CapabilityFunctionSuspend

If set to 1, indicates that the composite driver supports the USB function suspend feature.

Reserved

Reserved.

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 8.
Header	usbdlib.h

## See also

[COMPOSITE\\_DEVICE\\_CAPABILITIES\\_INIT](#)

[How to Register a Composite Device](#)

[IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

[REGISTER\\_COMPOSITE\\_DEVICE](#)

[USBD\\_BuildRegisterCompositeDevice](#)

# COMPOSITE\_DEVICE\_CAPABILITIES\_INIT function (usbdlib.h)

Article02/22/2024

The `COMPOSITE_DEVICE_CAPABILITIES_INIT` macro initializes the `COMPOSITE_DEVICE_CAPABILITIES` structure.

## Syntax

C++

```
void COMPOSITE_DEVICE_CAPABILITIES_INIT(
    PCOMPOSITE_DEVICE_CAPABILITIES CapabilityFlags
);
```

## Parameters

CapabilityFlags

A pointer to a caller-allocated `COMPOSITE_DEVICE_CAPABILITIES` structure to be initialized. The macro sets the `CompositeDriverCapabilityFunctionSuspend` member of `COMPOSITE_DEVICE_CAPABILITIES` to 0.

## Return value

None

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h

Requirement	Value
Library	Usbdex.lib
IRQL	<=DISPATCH_LEVEL

## See also

[COMPOSITE\\_DEVICE\\_CAPABILITIES](#)

[How to Register a Composite Device](#)

[REGISTER\\_COMPOSITE\\_DEVICE](#)

# GET\_ISO\_URB\_SIZE macro (usbdlib.h)

Article 02/22/2024

The **GET\_ISO\_URB\_SIZE** macro returns the number of bytes required to hold an isochronous transfer request.

## Syntax

C++

```
#define GET_ISO_URB_SIZE(n) (sizeof(struct _URB_ISOCH_TRANSFER)+\
    sizeof(USBD_ISO_PACKET_DESCRIPTOR)*n)
```

## Parameters

n

Specifies the number of isochronous transfer packets that will be part of the transfer request.

## Return value

None

## Remarks

Gets the number of bytes required to hold an isochronous request with the given number of packets (n).

## Requirements

[ ] Expand table

Requirement	Value
Target Platform	Windows
Header	usbdlib.h (include Usbdlib.h)

## See also

- [Routines for USB Client Drivers](#)
- [URB](#)
- [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#)
- [\\_URB\\_ISOCH\\_TRANSFER](#)

# REGISTER\_COMPOSITE\_DEVICE structure (usbdlib.h)

Article04/01/2021

The **REGISTER\_COMPOSITE\_DEVICE** structure is used with the [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#) I/O control request to register a parent driver of a Universal Serial Bus (USB) multi-function device (composite driver) with the USB driver stack.

To initialize the **REGISTER\_COMPOSITE\_DEVICE** structure, the composite driver must call the [USBD\\_BuildRegisterCompositeDevice](#) routine.

## Syntax

C++

```
typedef struct _REGISTER_COMPOSITE_DEVICE {
    USHORT          Version;
    USHORT          Size;
    USBDI_HANDLE   Reserved;
    COMPOSITE_DEVICE_CAPABILITIES CapabilityFlags;
    ULONG           FunctionCount;
} REGISTER_COMPOSITE_DEVICE, *PREGISTER_COMPOSITE_DEVICE;
```

## Members

**Version**

The version of this structure. [USBD\\_BuildRegisterCompositeDevice](#) sets this member.

**Size**

The size of this structure. [USBD\\_BuildRegisterCompositeDevice](#) sets this member.

**Reserved**

Reserved. [USBD\\_BuildRegisterCompositeDevice](#) sets this member.

**CapabilityFlags**

The capabilities that are supported by the composite driver. To specify that function suspend is supported by the composite driver, first initialize the

[COMPOSITE\\_DEVICE\\_CAPABILITIES](#) structure by calling the [COMPOSITE\\_DEVICE\\_CAPABILITIES\\_INIT](#) macro. Then, set the [CompositeDeviceCapabilityFunctionSuspend](#) member of [COMPOSITE\\_DEVICE\\_CAPABILITIES](#) to 1. Finally, call [USBD\\_BuildRegisterCompositeDevice](#) and pass the initialized structure in the *CapabilityFlags* parameter.

#### FunctionCount

The number of functions supported by the composite device. The **FunctionCount** value must not exceed 255.

## Requirements

Minimum supported client	Windows 8
Header	usbdlib.h

## See also

[COMPOSITE\\_DEVICE\\_CAPABILITIES](#)

[COMPOSITE\\_DEVICE\\_CAPABILITIES\\_INIT](#)

[IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

[USBD\\_BuildRegisterCompositeDevice](#)

# REQUEST\_REMOTE\_WAKE\_NOTIFICATION structure (usbdlib.h)

Article02/22/2024

The purpose of the **REQUEST\_REMOTE\_WAKE\_NOTIFICATION** structure is to specify input parameters for the [IOCTL\\_INTERNAL\\_USB\\_REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#) I/O control request.

## Syntax

C++

```
typedef struct _REQUEST_REMOTE_WAKE_NOTIFICATION {
    USHORT          Version;
    USHORT          Size;
    USBD_FUNCTION_HANDLE UsbdFunctionHandle;
    ULONG           Interface;
} REQUEST_REMOTE_WAKE_NOTIFICATION, *PREQUEST_REMOTE_WAKE_NOTIFICATION;
```

## Members

**Version**

The version of this structure. Set to 0.

**Size**

The size of the **REQUEST\_REMOTE\_WAKE\_NOTIFICATION** structure.

**UsbdFunctionHandle**

A function handle that is associated with the function that sends the resume signal. The handle was obtained in a previous [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#) request.

**Interface**

Specifies the device-defined index identifier of the interface with which the function is associated.

## Requirements

Requirement	Value
Minimum supported client	Windows 8
Header	usbdlib.h

## See also

[How to Implement Function Suspend in a Composite Driver](#)

[IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

[IOCTL\\_INTERNAL\\_USB\\_REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#)

# UsbBuildGetStatusRequest macro (usbdl.h)

Article 02/22/2024

The **UsbBuildGetStatusRequest** macro formats an [URB](#) to obtain status from a device, interface, endpoint, or other device-defined target on a USB device.

## Syntax

C++

```
void UsbBuildGetStatusRequest(
    [in, out]      urb,
    [in]           op,
    [in]           index,
    [in, optional] transferBuffer,
    [in, optional] transferBufferMDL,
    [in]           link
);
```

## Parameters

[in, out] urb

Pointer to an [URB](#) to be formatted as an status request.

[in] op

Specifies one of the following values:

### URB\_FUNCTION\_GET\_STATUS\_FROM\_DEVICE

Retrieves status from a USB device.

### URB\_FUNCTION\_GET\_STATUS\_FROM\_INTERFACE

Retrieves status from an interface on a USB device.

### URB\_FUNCTION\_GET\_STATUS\_FROM\_ENDPOINT

Retrieves status from an endpoint for an interface on a USB device.

## URB\_FUNCTION\_GET\_STATUS\_FROM\_OTHER

Retrieves status from a device-defined target on a USB device.

[in] `index`

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, *Index* must be zero.

[in, optional] `transferBuffer`

Pointer to a resident buffer to receive the status data or is **NULL** if an MDL is supplied in *TransferBufferMDL*.

[in, optional] `transferBufferMDL`

Pointer to an MDL that describes a resident buffer to receive the status data or is **NULL** if a buffer is supplied in *TransferBuffer*.

[in] `link`

Reserved. Must be set to **NULL**.

## Return value

None

## Requirements

  Expand table

Requirement	Value
Target Platform	Desktop
Header	usbdlib.h (include Usbdlib.h)

## See also

[URB](#)

## USB device driver programming reference

### \_URB\_CONTROL\_GET\_STATUS\_REQUEST

# UsbBuildInterruptOrBulkTransferRequest macro (usbdlib.h)

Article 10/21/2021

The **UsbBuildInterruptOrBulkTransferRequest** macro formats an [URB](#) to send or receive data on a bulk pipe, or to receive data from an interrupt pipe.

## Syntax

C++

```
void UsbBuildInterruptOrBulkTransferRequest(
    [in, out]          urb,
    [in]                length,
    [in]                pipeHandle,
    [in, optional]      transferBuffer,
    [in, optional]      transferBufferMDL,
    [in]                transferBufferLength,
    [in]                transferFlags,
    [in]                link
);
```

## Parameters

`[in, out] urb`

Pointer to an [URB](#) to be formatted as an interrupt or bulk transfer request.

`[in] length`

Specifies the size, in bytes, of the [URB](#).

`[in] pipeHandle`

Specifies the handle for this pipe returned by the HCD when a configuration was selected.

`[in, optional] transferBuffer`

Pointer to a resident buffer for the transfer or is **NULL** if an MDL is supplied in *TransferBufferMDL*. The contents of this buffer depend on the value of *TransferFlags*. If **USBD\_TRANSFER\_DIRECTION\_IN** is specified, this buffer will contain data read from the

device on return from the HCD. Otherwise, this buffer contains driver-supplied data to be transferred to the device.

[in, optional] `transferBufferMDL`

Pointer to an MDL that describes a resident buffer or is **NULL** if a buffer is supplied in *TransferBuffer*. The contents of the buffer depend on the value of *TransferFlags*. If `USBD_TRANSFER_DIRECTION_IN` is specified, the described buffer will contain data read from the device on return from the HCD. Otherwise, the buffer contains driver-supplied data to be transferred to the device. The MDL must be allocated from nonpaged pool.

[in] `transferBufferLength`

Specifies the length, in bytes, of the buffer specified in *TransferBuffer* or described in *TransferBufferMDL*.

[in] `transferFlags`

Specifies zero, one, or a combination of the following flags:

## **USBD\_TRANSFER\_DIRECTION\_IN**

Is set to request data from a device. To transfer data to a device, this flag must be clear.

## **USBD\_SHORT\_TRANSFER\_OK**

Can be used if `USBD_TRANSFER_DIRECTION_IN` is set. If set, directs the HCD not to return an error if a packet is received from the device that is shorter than the maximum packet size for the endpoint. Otherwise, a short request returns an error condition.

[in] `link`

Reserved. Must be set to **NULL**.

## **Return value**

None

## **Requirements**



Target Platform	Desktop
Header	usbdlib.h (include Usbdlib.h)

## See also

[URB](#)

[USB device driver programming reference](#)

[USB\\_DEVICE\\_DESCRIPTOR](#)

# UsbBuildOpenStaticStreamsRequest function (usbdlib.h)

Article 10/21/2021

The **UsbBuildOpenStaticStreamsRequest** inline function formats an [URB](#) structure for an open-streams request. The request opens streams associated with the specified bulk endpoint.

## Syntax

C++

```
void UsbBuildOpenStaticStreamsRequest(
    [in, out] PURB                      Urb,
    [in]      USBD_PIPE_HANDLE           PipeHandle,
    [in]      USHORT                   NumberOfStreams,
    [in]      PUSBD_STREAM_INFORMATION StreamInfoArray
);
```

## Parameters

[in, out] **Urb**

Pointer to the [URB](#) structure to be formatted for the open-stream request (URB\_FUNCTION\_OPEN\_STATIC\_STREAMS). The caller must allocate nonpaged pool for this [URB](#).

[in] **PipeHandle**

An opaque handle for the pipe associated with the endpoint that contains the streams to open.

The client driver obtains **PipeHandle** from a previous select-configuration request (URB\_FUNCTION\_SELECT\_CONFIGURATION) or a select-interface request (URB\_FUNCTION\_SELECT\_INTERFACE).

[in] **NumberOfStreams**

The number of streams to open. The **NumberOfStreams** value indicates the number of elements in the array pointed to by **Streams**. This value must be greater than zero and less than or equal to the maximum number of streams supported by the host controller

hardware. To get the maximum number of supported streams, call [USBD\\_QueryUsbCapability](#).

The number streams must also be less than or equal to the maximum number of streams supported by the USB device. To get that number, inspect the endpoint companion descriptor.

In the **NumberOfStreams** value, specify lesser of two values supported by the host controller and the USB device.

[in] `StreamInfoArray`

Pointer to a caller-allocated, initialized array of [USBD\\_STREAM\\_INFORMATION](#) structures. The length of the array depends on the number of streams to open and must be the same as the **NumberOfStreams** value.

## Return value

None

## Remarks

For a code example that shows the URB format required for an open-streams request, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h

## See also

[How to Open and Close Static Streams in a USB Bulk Endpoint](#)

[\\_URB\\_OPEN\\_STATIC\\_STREAMS](#)

# USBD\_AssignUrbToIoStackLocation function (usbdlib.h)

Article 10/21/2021

The **USBD\_AssignUrbToIoStackLocation** routine is called by a client driver to associate an [URB](#) with the IRP's next stack location.

## Syntax

C++

```
void USBD_AssignUrbToIoStackLocation(
    [in] USBD_HANDLE          USBDHandle,
    [in] PIO_STACK_LOCATION   IoStackLocation,
    [in] PURB                 Urb
);
```

## Parameters

[in] `USBDHandle`

A USBD handle that is retrieved in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] `IoStackLocation`

Pointer to the IRP's next stack location ([IO\\_STACK\\_LOCATION](#)). The client driver received a pointer to the stack location in a previous call to [IoGetNextIrpStackLocation](#).

[in] `Urb`

Pointer to the [URB](#) structure that is allocated by [USBD\\_UrbAllocate](#), [USBD\\_IsochUrbAllocate](#), [USBD\\_SelectConfigUrbAllocateAndBuild](#), or [USBD\\_SelectInterfaceUrbAllocateAndBuild](#).

## Return value

None

## Remarks

If the client driver allocated an URB by calling [USBD\\_UrbAllocate](#), [USBD\\_IsochUrbAllocate](#), [USBD\\_SelectConfigUrbAllocateAndBuild](#), or [USBD\\_SelectInterfaceUrbAllocateAndBuild](#), then the driver *must* call **USBD\_AssignUrbToloStackLocation** to associate the URB with [IO\\_STACK\\_LOCATION](#) associated with the IRP. For URBs that are allocated by those routines, **USBD\_AssignUrbToloStackLocation** replaces setting **Parameters.Others.Argument1** of [IO\\_STACK\\_LOCATION](#) to the URB. (see [IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_URB](#)).

The client driver must *not* call **USBD\_AssignUrbToloStackLocation** for an URB that is allocated by using other mechanisms, such as allocating the URB on the stack. Otherwise, the USB driver stack generates a bugcheck.

The client driver must call **USBD\_AssignUrbToloStackLocation** before calling [IoCallDriver](#) to send the request. **USBD\_AssignUrbToloStackLocation** populates the IRP's next stack location with the URB. The routine also updates the **FileObject** member of [IO\\_STACK\\_LOCATION](#).

For a code example, see [How to Submit an URB](#).

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdl.h
Library	Usbdex.lib
IRQL	<=DISPATCH_LEVEL

## See also

[How to Submit an URB](#)

[USBD\\_IsochUrbAllocate](#)

[USBD\\_SelectConfigUrbAllocateAndBuild](#)

[USBD\\_SelectInterfaceUrbAllocateAndBuild](#)

[USBD\\_UrbAllocate](#)

# USBD\_BuildRegisterCompositeDevice function (usbdlib.h)

Article 10/21/2021

The **USBD\_BuildRegisterCompositeDevice** routine is called by the driver of a USB multi-function device (composite driver) to initialize a **REGISTER\_COMPOSITE\_DEVICE** structure with the information required for registering the driver with the USB driver stack.

The routine is called by a driver that replaces the Microsoft-provided composite driver, Usbccgp.sys.

## Syntax

C++

```
void USBD_BuildRegisterCompositeDevice(
    [in] USBD_HANDLE             USBDHandle,
    [in] COMPOSITE_DEVICE_CAPABILITIES CapabilityFlags,
    [in] ULONG                   FunctionCount,
    [out] PREGISTER_COMPOSITE_DEVICE RegisterCompositeDevice
);
```

## Parameters

[in] **USBDHandle**

A USBD handle that is retrieved in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **CapabilityFlags**

A caller-allocated **COMPOSITE\_DEVICE\_CAPABILITIES** structure that indicates the capabilities that are supported by the composite driver. For instance, to indicate that the composite driver supports function suspend, set the **CapabilityFunctionSuspend** member of **COMPOSITE\_DEVICE\_CAPABILITIES** to 1.

[in] **FunctionCount**

The number of physical device objects (PDOs) to be created by the parent driver. The *FunctionCount* value cannot exceed 255.

[out] **RegisterCompositeDevice**

A pointer to a caller-allocated [REGISTER\\_COMPOSITE\\_DEVICE](#) structure. Upon completion, the structure is populated with the specified registration information. To register the composite driver, send the [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#) I/O request and pass the populated structure.

## Return value

None

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib
IRQL	< = DISPATCH_LEVEL

## See also

[IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

[REGISTER\\_COMPOSITE\\_DEVICE](#)

# USBD\_CalculateUsbBandwidth function (usbdl.h)

Article02/22/2024

The **USBD\_CalculateUsbBandwidth** routine has been deprecated in Windows XP and later operating systems. Do not use.

## Syntax

C++

```
ULONG USBD_CalculateUsbBandwidth(
    [in] ULONG    MaxPacketSize,
    [in] UCHAR    EndpointType,
    [in] BOOLEAN   LowSpeed
);
```

## Parameters

[in] **MaxPacketSize**

Specifies the maximum packet size.

[in] **EndpointType**

Contains a value of type **USBD\_PIPE\_TYPE** that specifies the pipe type.

[in] **LowSpeed**

Indicates, when **TRUE**, that the device is a low speed device. When **FALSE**, this member indicates that the device is a hi-speed device.

## Return value

The **USBD\_CalculateUsbBandwidth** routine returns zero for bulk and control endpoints and the bandwidth consumed in bits per millisecond. returns for all other endpoints.

## Remarks

The **USBD\_CalculateUsbBandwidth** routine approximates the bandwidth using the following procedure. First, **USBD\_CalculateUsbBandwidth** adds the largest possible packet size, specified in *MaxPacketSize*, to the overhead associated with the type of end point specified in *EndpointType*. Next, **USBD\_CalculateUsbBandwidth** multiplies this sum by 8 to convert the units from *bytes* per millisecond into *bits* per millisecond. Finally, **USBD\_CalculateUsbBandwidth** multiplies this quantity by 7/6 to account for filler bits. In a worst case scenario, there will be one bit of filler data stuffed into the data stream for every six bits of data. **USBD\_CalculateUsbBandwidth** uses worst-case assumptions to calculate the bandwidth required by the pipe.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Deprecated.
Target Platform	Universal
Header	usbdl.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	<=DISPATCH_LEVEL

## See also

[USB device driver programming reference](#)

[USBD\\_PIPE\\_TYPE](#)

# USBD\_CloseHandle function (usbdlib.h)

Article02/22/2024

The **USBD\_CloseHandle** routine is called by a USB client driver to close a USBD handle and release all resources associated with the driver's registration.

## Syntax

C++

```
void USBD_CloseHandle(
    [in] USBD_HANDLE USBDHandle
);
```

## Parameters

[in] **USBDHandle**

USBD handle to be closed. The handle is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

## Return value

None

## Remarks

A client driver should call **USBD\_CloseHandle** in the driver's routine that handles the [IRP\\_MN\\_REMOVE\\_DEVICE](#) IRP. The client driver must call the routine before sending the IRP down the USB driver stack.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Requires DDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.

Requirement	Value
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib
IRQL	PASSIVE_LEVEL

## See also

[Allocating and Building URBs](#)

[USBD\\_CreateHandle](#)

# USBD\_CreateConfigurationRequest function (usbdlib.h)

Article02/22/2024

The **USBD\_CreateConfigurationRequest** routine has been deprecated. Use [USBD\\_CreateConfigurationRequestEx](#) instead.

## Syntax

C++

```
PURB USBD_CreateConfigurationRequest(
    [in]      PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    [in, out] PUSHORT                  Siz
);
```

## Parameters

[in] ConfigurationDescriptor

Pointer to a caller-allocated [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure that contains the configuration descriptor for the configuration to be selected.

[in, out] Siz

Size of the [URB](#) structure.

## Return value

**USBD\_CreateConfigurationRequest** allocates a [URB](#) structure, formats it for the URB\_FUNCTION\_SELECT\_CONFIGURATION request (select-configuration request), and returns a pointer to the [URB](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Deprecated. Use USBD_CreateConfigurationRequestEx instead.
Target Platform	Universal
Header	usbdlib.h
Library	Usbd.lib

## See also

[USB device driver programming reference](#)

[USBD\\_CreateConfigurationRequestEx](#)

# USBD\_CreateConfigurationRequestEx function (usbdlib.h)

Article 10/21/2021

The **USBD\_CreateConfigurationRequestEx** routine allocates and formats a [URB](#) to select a configuration for a USB device.

**USBD\_CreateConfigurationRequestEx** replaces [USBD\\_CreateConfigurationRequest](#).

## Syntax

C++

```
PURB USBD_CreateConfigurationRequestEx(
    [in] PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    [in] PUSBD_INTERFACE_LIST_ENTRY     InterfaceList
);
```

## Parameters

[in] ConfigurationDescriptor

Pointer to a caller-allocated [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure that contains the configuration descriptor for the configuration to be selected. Typically, the client driver submits a URB of the type [URB\\_FUNCTION\\_GET\\_DESCRIPTOR\\_FROM\\_DEVICE](#) (see [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)) to retrieve information about configurations, interfaces, endpoints, the vendor, and class-specific descriptors from a USB device.

When the client driver specifies [USB\\_CONFIGURATION\\_DESCRIPTOR\\_TYPE](#) as the descriptor type, the request retrieves all device information in a [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure. The driver then passes the received pointer to the [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure in the *ConfigurationDescriptor* parameter.

[in] InterfaceList

Pointer to the first element of a caller-allocated array of [USBD\\_INTERFACE\\_LIST\\_ENTRY](#) structures. The length of the array depends on the number of interfaces in the configuration descriptor. The number of elements in the array must be one more than the number of interfaces in the configuration. Initialize the array by calling

[RtlZeroMemory](#). The **InterfaceDescriptor** member of the last element in the array must be set to **NULL**.

## Return value

**USBD\_CreateConfigurationRequestEx** allocates a **URB** structure, formats it for the **URB\_FUNCTION\_SELECT\_CONFIGURATION** request (select-configuration request), and returns a pointer to the **URB**. The client driver can then use the returned **URB** to send the select-configuration request to the host controller driver to set the configuration. You must free the **URB** when you have finished using it.

## Remarks

For information about how to build a select-configuration request and code example, see [How to Select a Configuration for a USB Device](#).

The returned value is a pointer to the **URB** structure that you can use to submit a select-configuration request to the host controller driver to set the specified configuration.

After the USB driver stack completes the select-configuration request, you can inspect the **USBD\_INTERFACE\_INFORMATION** structures. The **Pipes** member of **USBD\_INTERFACE\_INFORMATION** points to an array of **USBD\_PIPE\_INFORMATION** structures. The USB bus driver fills the array of **USBD\_PIPE\_INFORMATION** structures with information about the pipes associated with the endpoints of the interface. The client driver can obtain pipe handles from the `Pipes[i].PipeHandle` and use them to send I/O requests to specific pipes.

After you have completed all operations with the returned **URB**, you must free the **URB** by calling [ExFreePool](#).

You can allocate the configuration descriptor and the array from nonpaged or paged pool. Callers of this routine can run at IRQL <= DISPATCH\_LEVEL if the memory pointed to by *ConfigurationDescriptor* and *InterfaceList* is allocated from nonpaged pool.

Otherwise, callers must run at IRQL < DISPATCH\_LEVEL.

## Requirements

Target Platform	Universal

Header	usbdlib.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	DISPATCH_LEVEL (See Remarks)

## See also

[How to Select a Configuration for a USB Device](#)

[USB device driver programming reference](#)

[USBD\\_INTERFACE\\_INFORMATION](#)

[\\_URB\\_SELECT\\_CONFIGURATION](#)

# USBD\_CreateHandle function (usbdlib.h)

Article10/21/2021

The **USBD\_CreateHandle** routine is called by a WDM USB client driver to obtain a USBD handle. The routine registers the client driver with the underlying USB driver stack.

**Note for Windows Driver Framework (WDF) Drivers:** If your client driver is a WDF-based driver, then you do not need the USBD handle. The client driver is registered in its call to the [WdfUsbTargetDeviceCreateWithParameters](#) method.

## Syntax

C++

```
NTSTATUS USBD_CreateHandle(
    [in] PDEVICE_OBJECT DeviceObject,
    [in] PDEVICE_OBJECT TargetDeviceObject,
    [in] ULONG          USBDClientContractVersion,
    [in] ULONG          PoolTag,
    [out] USBD_HANDLE   *USBDHandle
);
```

## Parameters

[in] **DeviceObject**

Pointer to the device object for the client driver.

[in] **TargetDeviceObject**

Pointer to the next lower device object in the device stack. The client driver receives a pointer to that device object in a previous call to [IoAttachDeviceToDeviceStack](#).

[in] **USBDClientContractVersion**

The contract version that the client driver supports. *USBDClientContractVersion* must be **USBD\_CLIENT\_CONTRACT\_VERSION\_602**. For more information, see Remarks.

[in] **PoolTag**

The pool tag used for memory allocations.

[out] **USBDHandle**

Opaque handle that indicates that the client driver was registered with the USB driver stack. For more information, see Remarks.

## Return value

The routine returns an NTSTATUS code. Possible values include but are not limited to, these values in the following table.

Return code	Description
STATUS_SUCCESS	The routine call succeeded.
STATUS_INVALID_LEVEL	The caller is not running at the IRQL value PASSIVE_LEVEL.
STATUS_INVALID_PARAMETER	The caller passed one of the following invalid parameter values: <ul style="list-style-type: none"><li>• <i>DeviceObject</i>, <i>TargetDeviceObject</i>, or <i>USBDHandle</i> is NULL.</li><li>• The client contract value specified in <i>USBDClientContractVersion</i> is not valid.</li><li>• <i>PoolTag</i> is zero.</li></ul>

## Remarks

### Version Registration

Windows 8 includes a new USB driver stack to support USB 3.0 devices. The new USB driver stack provides several new capabilities, such as stream support, chained MDLs, and so on. Before your client driver can use any of those USB capabilities, you must register the client driver with the USB driver stack and obtain a USBD handle. The handle is required in order to call routines that use or configure the new capabilities. To obtain a USBD handle, call **USBD\_CreateHandle**.

The client driver must call **USBD\_CreateHandle** regardless of whether the device is attached to a USB 3.0, 2.0, or 1.1 host controller. If the device is attached to a USB 3.0 host controller, Windows loads the USB 3.0 driver stack. Otherwise, USB 2.0 driver stack is loaded. In either case, the client driver is *not* required to know the version supported by the underlying USB driver stack. **USBD\_CreateHandle** assesses the driver stack version and allocates resources appropriately.

The client driver must specify **USBD\_CLIENT\_CONTRACT\_VERSION\_602** in the *USBDClientContractVersion* parameter and follow the set of rules described in [Best](#)

## Calling USBD\_CreateHandle

The **USBD\_CreateHandle** routine must be called by a Windows Driver Model (WDM) client driver before the driver send any other requests, through URBs or IOCTLs, to the USB driver stack. Typically, the client driver obtains the USBD handle in its AddDevice routine.

A Windows Driver Frameworks (WDF) client driver is not required to call **USBD\_CreateHandle** because the framework calls this routine on behalf of the client driver during the device initialization phase. Instead, the client driver can specify its client contract version in the [WDF\\_USB\\_DEVICE\\_CREATE\\_CONFIG](#) structure and pass it in the call to [WdfUsbTargetDeviceCreateWithParameters](#).

## USBD\_CreateHandle Call Completion

If the **USBD\_CreateHandle** call succeeds, a valid *USBD handle* is obtained in the *USBDHandle* parameter. The client driver must use the USBD handle in the client driver's future requests to the USB driver stack.

If the **USBD\_CreateHandle** call fails, the client driver can fail the AddDevice routine.

After the client driver is finished using the USBD handle, the driver must close the handle by calling the [USBD\\_CloseHandle](#) routine.

## Examples

The following example code shows how to register a client driver by calling **USBD\_CreateHandle**.

C++

```
DRIVER_ADD_DEVICE MyAddDevice;

NTSTATUS MyAddDevice( __in PDRIVER_OBJECT  DriverObject,
                     __in PDEVICE_OBJECT  PhysicalDeviceObject)
{

    NTSTATUS          ntStatus;
    PDEVICE_OBJECT    deviceObject;
    PDEVICE_EXTENSION deviceExtension;
    PDEVICE_OBJECT    stackDeviceObject;
    USBD_HANDLE       usbdHandle;
```

```
...

    ntStatus = IoCreateDevice(DriverObject,
    sizeof(DEVICE_EXTENSION),
    NULL,
    FILE_DEVICE_UNKNOWN,
    FILE_AUTOGENERATED_DEVICE_NAME,
    FALSE,
    &deviceObject);

if (!NT_SUCCESS(ntStatus))
{
    return ntStatus;
}

...

//      Attach the FDO to the top of the PDO in the client driver's
//      device stack.

deviceExtension->StackDeviceObject = IoAttachDeviceToDeviceStack (
    deviceObject,
    PhysicalDeviceObject);

...

// Initialize the DeviceExtension

deviceExtension = deviceObject->DeviceExtension;

...

//Register the client driver with the USB driver stack.
//Obtain a USBD handle for registration.

ntStatus = USBD_CreateHandle(deviceObject,
    deviceExtension->StackDeviceObject,
    USBD_CLIENT_CONTRACT_VERSION_602,
    POOL_TAG,
    &deviceExtension->USBDHandle);

if (!NT_SUCCESS(ntStatus))
{
    return ntStatus;
}

...

// Call USBD_QueryUsbCapability to determine
// stream support.

ntStatus = USBD_QueryUsbCapability ( deviceExtension->USBDHandle,
    (GUID*)&GUID_USB_CAPABILITY_STATIC_STREAMS,
    sizeof(ULONG),
```

```
(PUCHAR) &deviceExtension.MaxSupportedStreams);  
  
    if (!NT_SUCCESS(ntStatus))  
    {  
        deviceExtension->MaxSupportedStreams = 0;  
        ntStatus = STATUS_SUCCESS;  
    }  
  
    ...  
}
```

## Requirements

<b>Minimum supported client</b>	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
<b>Target Platform</b>	Desktop
<b>Header</b>	usbdlib.h (include usbdlib.h, usb.h)
<b>Library</b>	Usbdex.lib; Ntstrsafe.lib
<b>IRQL</b>	PASSIVE_LEVEL

## See also

[Allocating and Building URBs](#)

[Best Practices: Using URBs](#)

[USBD\\_CloseHandle](#)

# USBD\_GetInterfaceLength function (usbdlib.h)

Article 02/22/2024

The **USBD\_GetInterfaceLength** routine obtains the length of a given interface descriptor, including the length of all endpoint descriptors contained within the interface.

## Syntax

C++

```
ULONG USBD_GetInterfaceLength(
    [in] PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor,
    [in] PUCHAR                 BufferEnd
);
```

## Parameters

[in] *InterfaceDescriptor*

Pointer to a interface descriptor for which to obtain the length.

[in] *BufferEnd*

Pointer to the position within a buffer at which to stop searching for the length of the interface and associated endpoints.

## Return value

**USBD\_GetInterfaceLength** returns the length, in bytes, of the interface descriptor and all associated endpoint descriptors contained within the interface.

## Remarks

Callers can use this routine to obtain the length of an interface and associated endpoints that are contained within another buffer. For example, a caller could specify a location inside of a larger buffer for *InterfaceDescriptor* and the beginning of a location of another interface descriptor for *BufferEnd*. This will cause the routine to search only

from the beginning of the interface descriptor specified by *InterfaceDescriptor* until either it finds another interface descriptor or it reaches the position specified by *BufferEnd*.

## Requirements

 Expand table

Requirement	Value
Target Platform	Universal
Header	usbdlib.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	PASSIVE_LEVEL

## See also

[USB device driver programming reference](#)

# USBD\_GetPdoRegistryParameter function (usbdlib.h)

Article02/22/2024

The **USBD\_GetPdoRegistryParameter** routine retrieves the value from the specified key in the USB device's hardware registry.

## Syntax

C++

```
NTSTATUS USBD_GetPdoRegistryParameter(
    [in]      PDEVICE_OBJECT PhysicalDeviceObject,
    [in, out] PVOID          Parameter,
    [in]      ULONG          ParameterLength,
    [in]      PWSTR           KeyName,
    [in]      ULONG          KeyNameLength
);
```

## Parameters

[in] **PhysicalDeviceObject**

Specifies the device object for the USB device.

[in, out] **Parameter**

Pointer to a caller-allocated buffer that receives the registry value.

[in] **ParameterLength**

Size, in bytes, of the buffer that is pointed to by *Parameter*.

[in] **KeyName**

Pointer to a string containing the name of the registry key.

[in] **KeyNameLength**

Size, in bytes, of the buffer that is pointed to by *KeyName*.

## Return value

The **USBD\_GetPdoRegistryParameter** returns STATUS\_SUCCESS when the operation succeeds or an appropriate error status when the operation fails.

## Requirements

 Expand table

Requirement	Value
Target Platform	Universal
Header	usbdlib.h
Library	Usbd.lib
IRQL	PASSIVE_LEVEL

## See also

[USB device driver programming reference](#)

# USBD\_GetUSBDIVersion function (usbdlib.h)

Article 02/22/2024

The **USBD\_GetUSBDIVersion** routine returns version information about the host controller driver (HCD) that controls the client's USB device.

**Note** **USBD\_IsInterfaceVersionSupported** replaces the **USBD\_GetUSBDIVersion** routine

## Syntax

C++

```
void USBD_GetUSBDIVersion(
    [out] PUSBD_VERSION_INFORMATION VersionInformation
);
```

## Parameters

[out] **VersionInformation**

Pointer to caller-allocated memory for a **USBD\_VERSION\_INFORMATION** structure that on return from the routine, contains version information about the HCD.

## Return value

None

## Remarks

Callers of this routine can be running at IRQL <= DISPATCH\_LEVEL if the memory for *VersionInformation* is allocated from nonpaged pool. Otherwise, callers must be running at IRQL < DISPATCH\_LEVEL.

# Requirements

 Expand table

Requirement	Value
Target Platform	Universal
Header	usbdlib.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	<=DISPATCH_LEVEL (See Remarks)

## See also

[USB device driver programming reference](#)

[USBD\\_IsInterfaceVersionSupported](#)

# USBD\_INTERFACE\_LIST\_ENTRY structure (usbdlib.h)

Article 02/22/2024

The **USBD\_INTERFACE\_LIST\_ENTRY** structure is used by USB client drivers to create an array of interfaces to be inserted into a configuration request.

## Syntax

C++

```
typedef struct _USBD_INTERFACE_LIST_ENTRY {
    PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor;
    PUSBD_INTERFACE_INFORMATION Interface;
} USBD_INTERFACE_LIST_ENTRY, *PUSBD_INTERFACE_LIST_ENTRY;
```

## Members

**InterfaceDescriptor**

Pointer to a [USB\\_INTERFACE\\_DESCRIPTOR](#) structure that describes the interface to be added to the configuration request.

**Interface**

Pointer to a [USBD\\_INTERFACE\\_INFORMATION](#) structure that describes the properties and settings of the interface pointed to by *InterfaceDescriptor*.

## Remarks

This structure is used by USB clients with the routine [USBD\\_CreateConfigurationRequestEx](#). Clients allocate an array of these structures, one for each interface to be configured. Clients must also allocate a NULL entry in the array to be used as a terminator before calling [USBD\\_CreateConfigurationRequestEx](#).

## Requirements

[] Expand table

Requirement	Value
Header	usbdlib.h (include Usbdlib.h)

## See also

[USB Structures](#)

[USBD\\_CreateConfigurationRequestEx](#)

# USBD\_IsInterfaceVersionSupported function (usbdlib.h)

Article02/22/2024

The **USBD\_IsInterfaceVersionSupported** routine is called by a USB client driver to check whether the underlying USB driver stack supports a particular USBD interface version.

## Syntax

C++

```
BOOLEAN USBD_IsInterfaceVersionSupported(
    [in] USBD_HANDLE USBDHandle,
    [in] ULONG      USBDInterfaceVersion
);
```

## Parameters

[in] `USBDHandle`

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] `USBDInterfaceVersion`

A LONG value that represents the USBD interface version to check against the USB driver stack. Possible values include `USBD_INTERFACE_VERSION_602` or `USBD_INTERFACE_VERSION_600`. For more information, see Remarks.

## Return value

**USBD\_IsInterfaceVersionSupported** returns TRUE if the specified USBD interface version is supported by the USB driver stack; FALSE otherwise. For more information, see Remarks.

## Remarks

The USB driver stack loaded for a device depends on the version of Windows, the host controller hardware, and the USB device. Windows 7 and earlier versions of Windows

support USBD\_INTERFACE\_VERSION\_600. The USBD interface versions, supported by the Windows 8 driver stack, are USBD\_INTERFACE\_VERSION\_602 and USBD\_INTERFACE\_VERSION\_600. A USB client driver rarely needs to know about the underlying driver stack's interface version. In cases where such information is required, the client driver can call the **USBD\_IsInterfaceVersionSupported** routine to check whether a particular interface version is supported by the underlying driver stack. For instance, the client driver calls **USBD\_IsInterfaceVersionSupported** to determine whether the driver stack supports USBD\_INTERFACE\_VERSION\_602. If it supports that version, the routine returns TRUE.

The routine requires a valid USBD handle (obtained in a previous call to [USBD\\_CreateHandle](#)). **USBD\_IsInterfaceVersionSupported** can only be called by client drivers that target Windows Vista and later versions of Windows. Those client drivers must get Windows Driver Kit (WDK) for Windows 8 in order to call the routines successfully. **USBD\_IsInterfaceVersionSupported** replaces the [USBD\\_GetUSBDiversion](#) routine.

The USBD interface version does not indicate the capabilities supported by the USB driver stack. For example just because the underlying driver stack supports USBD\_INTERFACE\_VERSION\_602, the client driver *must not* assume that the driver can use the static streams capability. That is because, even though the driver stack supports the capability, the host controller hardware or the USB device might not support streams. To determine whether the USB driver stack supports a certain capability, call [USBD\\_QueryUsbCapability](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbddlib.h
Library	Usbdex.lib
IRQL	PASSIVE_LEVEL

# USBD\_IsochUrbAllocate function (usbdlib.h)

Article03/16/2022

The **USBD\_IsochUrbAllocate** routine allocates and formats a [URB](#) structure for an isochronous transfer request.

## ⓘ Note

**For Windows Driver Framework (WDF) Drivers:** If your client driver is WDF-based, you must call the [WdfUsbTargetDeviceCreateIsochUrb](#) method instead of **USBD\_IsochUrbAllocate** to allocate memory for the URB structure.

## Syntax

C++

```
NTSTATUS USBD_IsochUrbAllocate(
    [in] USBD_HANDLE USBDHandle,
    [in] ULONG       NumberOfIsochPackets,
    [out] PURB       *Urb
);
```

## Parameters

[in] **USBDHandle**

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **NumberOfIsochPackets**

Specifies the maximum number of isochronous packets required to perform the transfer. The transfer buffer is described in a variable-length array of [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#) structures that stores information about each packet, such as byte offset of the packet within the buffer. The array is specified in the **IsoPacket** member of the [\\_URB\\_ISOCH\\_TRANSFER](#) structure, which is used to define the format of an isochronous request URB.

[out] `Urb`

Pointer to an [URB](#) structure, which receives the URB allocated by [USBD\\_IsochUrbAllocate](#). All members of the URB structure are set to zero. The allocated URB is large enough to hold the maximum number of isochronous packets indicated by [NumberOfIsochPacket](#).

The client driver must free the URB when the driver has finished using it by calling [USBD\\_UrbFree](#).

## Return value

The [USBD\\_IsochUrbAllocate](#) routine returns [STATUS\\_SUCCESS](#) if the request is successful. Otherwise, [USBD\\_UrbAllocate](#) sets *Urb* to `NULL` and returns an NT status failure code.

Possible values include, but are not limited to, [STATUS\\_INVALID\\_PARAMETER](#), which indicates the caller passed in `NULL` to *USBDHandle* or *Urb*.

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib
IRQL	<=DISPATCH_LEVEL

## See also

- [Allocating and Building URBs](#)
- [How to Transfer Data to USB Isochronous Endpoints](#)
- [USBD\\_ISO\\_PACKET\\_DESCRIPTOR](#)
- [\\_URB\\_ISOCH\\_TRANSFER](#)

# USBD\_ParseConfigurationDescriptor function (usbdlib.h)

Article02/22/2024

The **USBD\_ParseConfigurationDescriptor** routine has been deprecated. Use [USBD\\_ParseConfigurationDescriptorEx](#) instead.

## Syntax

C++

```
PUSB_INTERFACE_DESCRIPTOR USBD_ParseConfigurationDescriptor(  
    [in] PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,  
    [in] UCHAR InterfaceNumber,  
    [in] UCHAR AlternateSetting  
) ;
```

## Parameters

**[in] ConfigurationDescriptor**

Pointer to a USB configuration descriptor that contains the interface for which to search.

**[in] InterfaceNumber**

Specifies the device-defined index of the interface to be retrieved. This should be set to -1 if it should not be a search criterion.

**[in] AlternateSetting**

Specifies the device-defined alternate-setting index of the interface to be retrieved. If the caller does not wish the alternate setting value to be a search criterion, this parameter should be set to -1.

## Return value

**USBD\_ParseConfigurationDescriptor** returns a pointer to the first interface descriptor that matches the given search criteria. If no interface matches the search criteria, it returns **NULL**.

# Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Deprecated. Use USBD_ParseConfigurationDescriptorEx instead.
Target Platform	Universal
Header	usbdlib.h
Library	Usbd.lib

## See also

[USB device driver programming reference](#)

[USBD\\_ParseConfigurationDescriptorEx](#)

# USBD\_ParseConfigurationDescriptorEx function (usbdlib.h)

Article 10/21/2021

The **USBD\_ParseConfigurationDescriptorEx** routine searches a given configuration descriptor and returns a pointer to an interface that matches the given search criteria.

## Syntax

C++

```
PUSB_INTERFACE_DESCRIPTOR USBD_ParseConfigurationDescriptorEx(
    [in] PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    [in] PVOID StartPosition,
    [in] LONG InterfaceNumber,
    [in] LONG AlternateSetting,
    [in] LONG InterfaceClass,
    [in] LONG InterfaceSubClass,
    [in] LONG InterfaceProtocol
);
```

## Parameters

[in] ConfigurationDescriptor

Pointer to a USB configuration descriptor that contains the interface for which to search.

[in] StartPosition

Pointer to the address within the configuration descriptor, provided at *ConfigurationDescriptor*, to begin searching from. To search from the beginning of the configuration descriptor, the parameters *ConfigurationDescriptor* and *StartPosition* must be the same address.

[in] InterfaceNumber

Specifies the device-defined index of the interface to be retrieved. This should be set to -1 if it should not be a search criterion.

[in] AlternateSetting

Specifies the device-defined alternate-setting index of the interface to be retrieved. If the caller does not wish the alternate setting value to be a search criterion, this parameter should be set to -1.

[in] InterfaceClass

Specifies the device- or USB-defined identifier for the interface class of the interface to be retrieved. If the caller does not wish the interface class value to be a search criterion, this parameter should be set to -1.

[in] InterfaceSubClass

Specifies the device- or USB-defined identifier for the interface subclass of the interface to be retrieved. If the caller does not wish the interface subclass value to be a search criterion, this parameter should be set to -1.

[in] InterfaceProtocol

Specifies the device- or USB-defined identifier for the interface protocol of the interface to be retrieved. If the caller does not wish the interface protocol value to be a search criterion, this parameter should be set to -1.

## Return value

**USBD\_ParseConfigurationDescriptorEx** returns a pointer to the first interface descriptor that matches the given search criteria. If no interface matches the search criteria, it returns **NULL**.

## Remarks

Callers can specify more than one of the search criteria (InterfaceNumber, AlternateSetting, InterfaceClass, InterfaceSubClass, and InterfaceProtocol) when using this routine to find an interface within a configuration descriptor. For example code, see [USBD\\_CreateConfigurationRequestEx](#).

When this routine parses the configuration descriptor looking for the interface descriptor that matches the search criteria, it returns the first match, terminating the search. Callers should specify as many search criteria as are necessary to find the desired interface.

## Requirements

Target Platform	Universal
Header	usbdlib.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	< DISPATCH_LEVEL

## See also

[USB device driver programming reference](#)

[USB\\_CONFIGURATION\\_DESCRIPTOR](#)

# USBD\_ParseDescriptors function (usbdl.h)

Article02/22/2024

The **USBD\_ParseDescriptors** routine searches a given configuration descriptor and returns a pointer to the first descriptor that matches the search criteria.

## Syntax

C++

```
PUSB_COMMON_DESCRIPTOR USBD_ParseDescriptors(
    [in] PVOID DescriptorBuffer,
    [in] ULONG TotalLength,
    [in] PVOID StartPosition,
    [in] LONG DescriptorType
);
```

## Parameters

[in] **DescriptorBuffer**

Pointer to a configuration descriptor that contains the descriptor for which to search.

[in] **TotalLength**

Specifies the size, in bytes, of the buffer pointed to by *DescriptorBuffer*.

[in] **StartPosition**

Pointer to the address within the configuration descriptor, provided at *DescriptorBuffer*, to begin searching from. To search from the beginning of the configuration descriptor, the parameters *DescriptorBuffer* and *StartPosition* must be the same address.

[in] **DescriptorType**

Specifies the descriptor type code as assigned by USB. The following values are valid for USB-defined descriptor types:

**USB\_STRING\_DESCRIPTOR\_TYPE**

Specifies that the descriptor being searched for is a string descriptor.

## USB\_INTERFACE\_DESCRIPTOR\_TYPE

Specifies that the descriptor being searched for is an interface descriptor.

## USB\_ENDPOINT\_DESCRIPTOR\_TYPE

Specifies that the descriptor being searched for is an endpoint descriptor.

## Return value

**USBD\_ParseDescriptors** returns a pointer to a [USB\\_COMMON\\_DESCRIPTOR](#) structure that is the head of the first descriptor that matches the given search criteria, or **NULL** is returned if no match is found:

## Remarks

This structure is used to hold a portion of a descriptor, so that the caller of **USBD\_ParseDescriptors** can determine the correct structure to use to access the remaining data in the descriptor. Every descriptor type has these fields at the beginning of the data and callers can use the **bLength** and **bDescriptorType** members to correctly identify the type of this descriptor.

When this routine parses the configuration descriptor looking for the descriptor that matches the search criteria, it returns the first match, terminating the search.

## Requirements

[+] Expand table

Requirement	Value
Target Platform	Universal
Header	usbdl.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	< DISPATCH_LEVEL

## See also

[USB device driver programming reference](#)

# USBD\_QueryBusTime function (usbdl.h)

Article02/22/2024

The **USBD\_QueryBusTime** routine has been deprecated in Windows XP and later operating systems. Do not use.

See URB\_FUNCTION\_GET\_CURRENT\_FRAME\_NUMBER for equivalent functionality that is supported on all versions of Windows.

## Syntax

C++

```
NTSTATUS USBD_QueryBusTime(
    [in] PDEVICE_OBJECT RootHubPdo,
    [out] PULONG           CurrentFrame
);
```

## Parameters

[in] RootHubPdo

Obsolete.

[out] CurrentFrame

Obsolete.

## Return value

This routine does not return a value.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Deprecated.

Requirement	Value
Target Platform	Universal
Header	usbdlib.h
Library	Usbd.lib

## See also

[URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#)

[USB device driver programming reference](#)

# USBD\_QueryUsbCapability function (usbdl.h)

Article 10/21/2021

The **USBD\_QueryUsbCapability** routine is called by a WDM client driver to determine whether the underlying USB driver stack and the host controller hardware support a specific capability. **Note for Windows Driver Framework (WDF) Drivers:** If your client driver is a WDF-based driver, then you must call the [WdfUsbTargetDeviceQueryUsbCapability](#) method instead of **USBD\_QueryUsbCapability**.

## Syntax

C++

```
NTSTATUS USBD_QueryUsbCapability(
    [in]          USBD_HANDLE USBDHandle,
    [in]          const GUID *CapabilityType,
    [in]          ULONG     OutputBufferLength,
    [in, out]      P UCHAR   OutputBuffer,
    [out, optional] P ULONG   ResultLength
);
```

## Parameters

[in] **USBDHandle**

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **CapabilityType**

Pointer to a GUID that represents the capability for which the client driver wants to retrieve information. The possible *PGUID* values are as follows:

- **GUID\_USB\_CAPABILITY\_CHAINED\_MDLS**
- **GUID\_USB\_CAPABILITY\_STATIC\_STREAMS**
- **GUID\_USB\_CAPABILITY\_SELECTIVE\_SUSPEND**
- **GUID\_USB\_CAPABILITY\_FUNCTION\_SUSPEND**
- **GUID\_USB\_CAPABILITY\_DEVICE\_CONNECTION\_HIGH\_SPEED\_COMPATIBLE**
- **GUID\_USB\_CAPABILITY\_DEVICE\_CONNECTION\_SUPER\_SPEED\_COMPATIBLE**
- **GUID\_USB\_CAPABILITY\_TIME\_SYNC**

[in] *OutputBufferLength*

Length, in bytes, of the buffer pointed to by *OutputBuffer*.

[in, out] *OutputBuffer*

Pointer to a caller-allocated buffer. Certain capability requests return additional information in an output buffer. For those requests, you must allocate the buffer and provide a pointer to the buffer in the *OutputBuffer* parameter. Currently, only the static-streams capability request requires an output buffer of the type USHORT. The buffer is filled by **USBD\_QueryUsbCapability** with the maximum number of streams supported per endpoint.

Other capability requests do not require an output buffer. For those requests, you must set *OutputBuffer* to NULL and *OutputBufferLength* to 0.

[out, optional] *ResultLength*

Pointer to a ULONG variable that receives the actual number of bytes in the buffer pointed to by *OutputBuffer*. The caller can pass NULL in *ResultLength*. If *ResultLength* is not NULL, the received value is less than or equal to the *OutputBufferLength* value.

## Return value

The **USBD\_QueryUsbCapability** routine returns an NT status code.

Possible values include, but are not limited to, the status codes listed in the following table.

Return code	Description
STATUS_SUCCESS	The request was successful and the specified capability is supported.
STATUS_INVALID_PARAMETER	The caller passed an invalid parameter value. <ul style="list-style-type: none"><li>• <i>USBDHandle</i> or <i>CapabilityType</i> is NULL.</li><li>• <i>OutputBuffer</i> is NULL but <i>OutputBufferLength</i> indicates a nonzero value. Conversely, the caller provided an output buffer but the buffer length is 0.</li></ul>
STATUS_NOT_IMPLEMENTED	The specified capability is not supported by the underlying USB driver stack.
STATUS_NOT_SUPPORTED	The specified capability is not supported either by the host controller hardware or the USB driver stack.

## Remarks

Windows 8 includes a new USB driver stack to support USB 3.0 devices. The new USB driver stack provides several new capabilities defined such as, stream support and chained MDLs that can be used by a client driver.

A client driver can determine the version of the underlying USB driver stack by calling the [IsInterfaceVersionSupported](#) routine.

The client driver can use the new capabilities *only if* the underlying USB driver stack *and* hardware support them. For example, in order to send I/O requests to a particular stream associated with a bulk endpoint, the underlying USB driver stack, the endpoint, and the host controller hardware must support the static streams capability. The client driver *must not* call [IsInterfaceVersionSupported](#) and assume the capabilities of the driver stack. Instead, the client driver *must* always call [USBD\\_QueryUsbCapability](#) to determine whether the USB driver stack and hardware support a particular capability.

The following table describes the USB-specific capabilities that a client driver can query through a [USBD\\_QueryUsbCapability](#) call.

Capability GUID	Description
GUID_USB_CAPABILITY_CHAINED_MDLS	<p>If the USB driver stack supports chained MDLs, the client driver can provide the transfer data as a chain of MDLs that reference segmented buffers in physical memory. For more information, see <a href="#">MDL</a>. Chained MDLs preclude the need for allocating and copying memory to create virtually contiguous buffers and therefore make I/O transfers more efficient. For more information, see <a href="#">How to Send Chained MDLs</a>.</p>
GUID_USB_CAPABILITY_STATIC_STREAMS	<p>If supported, the client driver can send I/O requests to streams in a bulk endpoint. For the static streams query request, the client driver is</p>

required to provide an output buffer (USHORT). After the call completes and if the static streams capability is supported, the output buffer receives the maximum number of supported streams by the host controller.

The output buffer value does not indicate the maximum number of streams supported by the bulk endpoint in the device. To determine that number, the client driver must inspect the endpoint companion descriptor.

The USB driver stack in Windows 8 supports up to 255 streams.

If static streams are supported, the client driver can send I/O requests to the first stream (also called the *default stream*) by using the pipe handle obtained through a select-configuration request. For other streams in the endpoint, the client driver must open those streams and obtain pipe handles for them in order to send I/O requests. For more information about opening streams, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

#### GUID\_USB\_CAPABILITY\_FUNCTION\_SUSPEND

This capability determines whether the underlying USB driver stack supports USB Function Suspend and Remote Wake-Up features. If supported, the driver stack can process a resume

signal (for remote wake-up) from an individual function in a USB 3.0 composite device. Based on that signal, an individual function driver can exit the low-power state of its function.

The capability is intended to be used by a composite driver: the driver that is loaded as the function device object (FDO) in the device stack for the composite device. By default, the Microsoft-provided USB Generic Parent Driver (Usbccgp.sys) is loaded as the FDO.

If your driver replaces Usbccgp.sys, the driver must be able to request remote wake-up and propagate the resume signal from the USB driver stack. Before implementing that logic, the driver must determine the USB driver stack's support for the function suspend capability by calling **USBD\_QueryUsbCapability**. Usbccgp.sys in Windows 8 implements function suspend.

For a code example and more information about function suspend, see [How to Implement Function Suspend in a Composite Driver](#).

#### GUID\_USB\_CAPABILITY\_SELECTIVE\_SUSPEND

Determines whether the underlying USB driver stack supports selective suspend. For information about selective suspend, see [USB Selective Suspend](#).

GUID_USB_CAPABILITY_DEVICE_CONNECTION_HIGH_SPEED_COMPATIBLE	Determines whether the bus is operating at high-speed or higher.
GUID_USB_CAPABILITY_DEVICE_CONNECTION_SUPER_SPEED_COMPATIBLE	Determines whether the bus is operating at SuperSpeed or higher.
GUID_USB_CAPABILITY_TIME_SYNC	Determines whether the frame number and QPC association feature is supported on the controller.

## Examples

The code snippet shows how to call **USBD\_QueryUsbCapability** to determine the capabilities of the underlying USB driver stack.

C++

```
/*++

Routine Description:
  This helper routine queries the underlying USB driver stack
  for specific capabilities. This code snippet assumes that
  USBD handle was retrieved by the client driver in a
  previous call to the USBD_CreateHandle routine.

Parameters:
  fdo: Pointer to the device object that is the current top
       of the stack as reported by IoAttachDeviceToDeviceStack.

Return Value: VOID
--*/

VOID QueryUsbDriverStackCaps (PDEVICE_OBJECT fdo)
{
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PDEVICE_EXTENSION deviceExtension;

    deviceExtension = (PDEVICE_EXTENSION)fdo->DeviceExtension;

    if (!deviceExtension->UsbdHandle)
    {
        return;
    }
```

```

// Check if the underlying USB driver stack
// supports USB 3.0 devices.

if (!USBD_IsInterfaceVersionSupported(
    deviceExtension->UsbdHandle,
    USBD_INTERFACE_VERSION_602))
{
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Old USB stack
loaded.\n" ));
}
else
{
    // Call USBD_QueryUsbCapability to determine
    // function suspend support.
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "New USB stack
loaded.\n" ));
    ntStatus = USBD_QueryUsbCapability ( deviceExtension->UsbdHandle,
        (GUID*)&GUID_USB_CAPABILITY_FUNCTION_SUSPEND,
        0,
        NULL,
        NULL);

    if (NT_SUCCESS(ntStatus))
    {
        deviceExtension->FunctionSuspendSupported = TRUE;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Function
suspend supported.\n" ));
    }
    else
    {
        deviceExtension->FunctionSuspendSupported = FALSE;
        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Function
suspend not supported.\n" ));
    }
}

// Call USBD_QueryUsbCapability to determine
// chained MDL support.

ntStatus = USBD_QueryUsbCapability(
    deviceExtension->UsbdHandle,
    (GUID*)&GUID_USB_CAPABILITY_CHAINED_MDLS,
    0,
    NULL,
    NULL);

if (NT_SUCCESS(ntStatus))
{
    deviceExtension->ChainedMDLSupport = TRUE;
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Chained MDLs
supported.\n" ));
}
else
{

```

```

        deviceExtension->ChainedMDLSupport = FALSE;
        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Chained MDLs
not supported.\n" ));
    }

    // Call USBD_QueryUsbCapability to determine
    // stream support.

    ntStatus = USBD_QueryUsbCapability (deviceExtension->UsbdHandle,
        (GUID*)&GUID_USB_CAPABILITY_STATIC_STREAMS,
        sizeof(ULONG),
        (PUCHAR) &deviceExtension->MaxSupportedStreams,
        NULL);

    if (!NT_SUCCESS(ntStatus))
    {
        deviceExtension->MaxSupportedStreams = 0;
        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Static streams
not supported.\n" ));
    }

    // Call USBD_QueryUsbCapability to determine
    // selective suspend support.

    ntStatus = USBD_QueryUsbCapability (deviceExtension->UsbdHandle,
        (GUID*)&GUID_USB_CAPABILITY_SELECTIVE_SUSPEND,
        0,
        NULL,
        NULL);

    if (!NT_SUCCESS(ntStatus))
    {
        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Selective
suspend not supported.\n" ));
    }
    else
    {
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "Selective
suspend supported.\n" ));
    }

    // Call USBD_QueryUsbCapability to determine
    // device speed.

    ntStatus = USBD_QueryUsbCapability (deviceExtension->UsbdHandle,
        (GUID*)&GUID_USB_CAPABILITY_DEVICE_CONNECTION_HIGH_SPEED_COMPATIBLE,
        0,
        NULL,
        NULL);

    if (!NT_SUCCESS(ntStatus))
    {

```

```

        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "The device is
operating at full speed or lower.\n The device can operate at high speed or
higher." ));
    }
    else
    {
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "The device is
operating at high speed or higher.\n" ));
    }

    // Call USBD_QueryUsbCapability to determine
    // device speed.
    ntStatus = USBD_QueryUsbCapability (deviceExtension->UsbdHandle,
(GUID*)&GUID_USB_CAPABILITY_DEVICE_CONNECTION_SUPER_SPEED_COMPATIBLE,
    0,
    NULL,
    NULL);

    if (!NT_SUCCESS(ntStatus))
    {
        ntStatus = STATUS_SUCCESS;
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "The device is
operating at high speed or lower.\n The device can operate at Superspeed or
higher." ));
    }
    else
    {
        KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "The device is
operating at SuperSpeed or higher.\n" ));
    }

    return;
}

}

```

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h (include Usbdlib.h)
Library	Usbdex.lib

IRQL

PASSIVE\_LEVEL

## See also

[USB device driver programming reference](#)

# USBD\_RegisterHcFilter function (usbdlib.h)

Article02/22/2024

The **USBD\_RegisterHcFilter** routine has been deprecated in Windows XP and later operating systems. Do not use.

On Windows XP and later operating systems, a filter driver that is installed between the root hub FDO and PDO sees all USB traffic for a USB device after it has been enumerated. There is no supported mechanism for filtering descriptor requests that occur during the enumeration of a USB device, because those requests originate and remain in the port driver (usbport.sys) and not the hub driver.

## Syntax

C++

```
void USBD_RegisterHcFilter(
    [in] PDEVICE_OBJECT DeviceObject,
    [in] PDEVICE_OBJECT FilterDeviceObject
);
```

## Parameters

[in] `DeviceObject`

Pointer to the device object that is the current top of the stack as reported by [IoAttachDeviceToDeviceStack](#).

[in] `FilterDeviceObject`

Pointer to the filter device object created by the filter driver for its operations.

## Return value

None

## Remarks

USB bus filter drivers must call this routine after attaching their device object to the device object stack for the host controller driver.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Deprecated.
Target Platform	Universal
Header	usbdl.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	PASSIVE_LEVEL

## See also

[IoAttachDeviceToDeviceStack](#)

[USB device driver programming reference](#)

# USBD\_SelectConfigUrbAllocateAndBuild function (usbdlib.h)

Article 10/21/2021

The **USBD\_SelectConfigUrbAllocateAndBuild** routine allocates and formats a [URB](#) structure that is required to select a configuration for a USB device.

**Note** In Windows 8, **USBD\_SelectConfigUrbAllocateAndBuild** replaces **USBD\_CreateConfigurationRequestEx** and **USBD\_CreateConfigurationRequest**.

## Syntax

C++

```
NTSTATUS USBD_SelectConfigUrbAllocateAndBuild(
    [in] USBD_HANDLE             USBDHandle,
    [in] PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    [in] PUSB_INTERFACE_LIST_ENTRY InterfaceList,
    [out] PURB                  *Urb
);
```

## Parameters

[in] **USBDHandle**

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **ConfigurationDescriptor**

Pointer to a caller-allocated [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure that contains the configuration descriptor for the configuration to be selected. Typically, the client driver submits an URB of the type [URB\\_FUNCTION\\_GET\\_DESCRIPTOR\\_FROM\\_DEVICE](#) (see [\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)) to retrieve information about configurations, interfaces, endpoints, the vendor, and class-specific descriptors from a USB device. When the client driver specifies [USB\\_CONFIGURATION\\_DESCRIPTOR\\_TYPE](#) as the descriptor type, the request retrieves all device information in a [USB\\_CONFIGURATION\\_DESCRIPTOR](#) structure. The driver then passes the received

pointer to the **USB\_CONFIGURATION\_DESCRIPTOR** structure in the *ConfigurationDescriptor* parameter.

[in] **InterfaceList**

Pointer to the first element of a caller-allocated array of **USBD\_INTERFACE\_LIST\_ENTRY** structures. The length of the array depends on the number of interfaces in the configuration descriptor. For more information, see Remarks.

[out] **Urb**

Pointer to a **URB** structure that receives the URB allocated by **USBD\_SelectConfigUrbAllocateAndBuild**. The client driver must free the URB when the driver has finished using it by calling **USBD\_UrbFree**.

## Return value

The **USBD\_SelectConfigUrbAllocateAndBuild** routine returns an NT status code.

Possible values include, but are not limited to, the status codes listed in the following table.

Return code	Description
<b>STATUS_SUCCESS</b>	The request was successful.
<b>STATUS_INVALID_PARAMETER</b>	The caller passed an invalid parameter value. <i>USBDHandle</i> or <i>Urb</i> is NULL.
<b>STATUS_INSUFFICIENT_RESOURCES</b>	Insufficient memory available to complete the request.

## Remarks

Before calling **USBD\_SelectConfigUrbAllocateAndBuild**, the client driver must perform the following tasks:

1. Get the number of interfaces in the configuration. This information is contained in the **bNumInterfaces** member of the **USB\_CONFIGURATION\_DESCRIPTOR** structure pointed to by *ConfigurationDescriptor*.
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**.

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](#).
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.

**USBD\_SelectConfigUrbAllocateAndBuild** 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.
- Fills a [USBD\\_INTERFACE\\_INFORMATION](#) structure in the URB for each interface.
- 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.

You can use the received pointer to the [URB](#) structure to submit a select-configuration request to the USB driver stack to set the specified configuration. In addition, you can use the **Interface** member of each [USBD\\_INTERFACE\\_INFORMATION](#) structure in the array to get information about the interface. Within each [USBD\\_INTERFACE\\_INFORMATION](#) structure, the **Pipes** member is an array of [USBD\\_PIPE\\_INFORMATION](#) structures. Each [USBD\\_PIPE\\_INFORMATION](#) structure contains information about the pipes opened (by the USB driver stack) for the endpoints in that interface. You can obtain pipe handles from the array and store them for future I/O requests to the device.

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib

IRQL

DISPATCH\_LEVEL

## See also

[USBD\\_CreateConfigurationRequestEx](#)

[USBD\\_CreateHandle](#)

# USBD\_SelectInterfaceUrbAllocateAndBuild function (usbdlib.h)

Article 10/21/2021

The **USBD\_SelectInterfaceUrbAllocateAndBuild** routine allocates and formats a [URB](#) structure that is required for a request to select an interface or change its alternate setting.

## Syntax

C++

```
NTSTATUS USBD_SelectInterfaceUrbAllocateAndBuild(
    [in] USBD_HANDLE             USBDHandle,
    [in] USBD_CONFIGURATION_HANDLE ConfigurationHandle,
    PUSBD_INTERFACE_LIST_ENTRY InterfaceListEntry,
    [out] PURB                  *Urb
);
```

## Parameters

[in] **USBDHandle**

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **ConfigurationHandle**

Handle returned by the USB driver stack in the **UrbSelectConfiguration.ConfigurationHandle** member of the [URB](#) structure, after the driver stack completes a select-configuration request.

**InterfaceListEntry**

Pointer to a caller-allocated [USBD\\_INTERFACE\\_LIST\\_ENTRY](#) structure. For more information, see Remarks.

[out] **Urb**

Pointer to a [URB](#) structure that receives the URB allocated by **USBD\_SelectInterfaceUrbAllocateAndBuild**. The client driver must free the URB when

the driver has finished using it by calling [USBD\\_UrbFree](#).

## Return value

The routine returns an NTSTATUS code. Possible values include but are not limited to, the status codes listed in the following table.

Return code	Description
STATUS_SUCCESS	The routine call succeeded.
STATUS_INVALID_PARAMETER	The caller passed NULL in any of the parameters.
STATUS_INSUFFICIENT_RESOURCES	Insufficient memory available to complete the call.

## Remarks

The client driver must call the [USBD\\_SelectInterfaceUrbAllocateAndBuild](#) routine after selecting a configuration in the device. After a select-configuration request completes, the client driver receives a configuration handle in the [UrbSelectConfiguration.ConfigurationHandle](#) member of the URB. That handle must be specified in the *ConfigurationHandle* parameter of [USBD\\_SelectInterfaceUrbAllocateAndBuild](#).

A client driver calls [USBD\\_SelectInterfaceUrbAllocateAndBuild](#) to allocate and build an URB for a select-interface request to change the alternate setting of an interface, in the selected configuration. In the call to [USBD\\_SelectInterfaceUrbAllocateAndBuild](#), the client driver must allocate and provide a pointer to a [USBD\\_INTERFACE\\_LIST\\_ENTRY](#) structure. The client driver must set the structure members as follows:

- The **InterfaceDescriptor** member must point to a [USB\\_INTERFACE\\_DESCRIPTOR](#) structure that contains the interface descriptor with the alternate setting to select. The interface descriptor was obtained in a previous request to get a configuration descriptor and the associated interface and endpoint descriptors.
- The **Interface** member must be NULL.

[USBD\\_SelectInterfaceUrbAllocateAndBuild](#) allocates an [URB](#) structure and fills it with information about the specified interface setting, and endpoints. The routine also allocates a [USBD\\_INTERFACE\\_INFORMATION](#) structure. The structure members (except pipe information) are filled based on the specified interface descriptor. [USBD\\_SelectInterfaceUrbAllocateAndBuild](#) sets the **Interface** member of [USBD\\_INTERFACE\\_LIST\\_ENTRY](#) to the address of [USBD\\_INTERFACE\\_INFORMATION](#) in

the URB. The client driver can send this URB to the USB driver stack to select an alternate setting in the interface.

A client driver cannot change alternate settings in multiple interfaces in a single select-interface request. Each request targets only one interface.

After the select-interface request is complete, the USB driver stack populates [USBD\\_INTERFACE\\_INFORMATION](#) with information about pipes opened for endpoints that are defined in the selected alternate setting. The client driver can obtain those pipe handles by inspecting the array pointed to by the **Pipes** member of [USBD\\_INTERFACE\\_INFORMATION](#), and store the handles for future data transfer requests.

The client driver can reuse an URB allocated by [USBD\\_SelectInterfaceUrbAllocateAndBuild](#) *only* for another select-interface request for the same alternate setting. The client driver *must not* reuse the URB for any other type of request, or for another select-interface request for a different alternate setting. Instead of allocating a new URB, reusing an existing URB is the preferred approach in certain scenarios. Consider a USB audio device that has an interface with two alternate settings, defined for two bandwidth requirements. Setting 0 is defined for zero bandwidth; Setting 1 is defined to use a certain amount of bandwidth. The client driver wants to frequently switch between the two settings depending on whether the device is in use. To implement this scenario, the client driver can allocate two URBs for select-interface requests, one per setting. The client driver can use (and reuse) an URB for a select-interface request to select Setting 1 when there are sounds to send to the device. To conserve bandwidth when there are no sounds, the client driver can use (and reuse) the other URB to switch to Setting 0. This implementation prevents the client driver from allocating URBs for each of those select-interface requests, every time the driver needs to change the setting.

## Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib
IRQL	DISPATCH_LEVEL

## See also

[USBD\\_CreateHandle](#)

[USBD\\_SelectConfigUrbAllocateAndBuild](#)

# USBD\_UrbAllocate function (usbdlib.h)

Article10/21/2021

The **USBD\_UrbAllocate** routine allocates a USB Request Block (URB).

## Syntax

C++

```
NTSTATUS USBD_UrbAllocate(
    [in] USBD_HANDLE USBDHandle,
    [out] PURB      *Urb
);
```

## Parameters

[in] `USBDHandle`

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[out] `Urb`

Pointer to the newly allocated [URB](#) structure. All members of the structure are set to zero. The client driver must free the URB when the driver has finished using it by calling [USBD\\_UrbFree](#).

## Return value

The **USBD\_UrbAllocate** routine returns `STATUS_SUCCESS` if the request is successful. Otherwise, **USBD\_UrbAllocate** sets *Urb* to `NULL` and returns a failure code.

Possible values include, but are not limited to, `STATUS_INVALID_PARAMETER`, which indicates the caller passed in `NULL` to *USBDHandle* or *Urb*.

## Remarks

The **USBD\_UrbAllocate** routine enables the underlying USB driver stack to allocate an opaque URB context for the URB. By using the URB context, the USB driver stack can process requests more efficiently and reliably. Those optimizations are provided by the

USB 3.0 driver stack that is included in Windows 8. The client driver cannot access the URB context; the context is used internally by the bus driver.

Regardless of the USB protocol version of the host controller, the underlying USB driver stack, the target operating system, the client driver must always call **USBD\_UrbAllocate** to allocate an [URB](#) structure. **USBD\_UrbAllocate** replaces earlier allocation mechanisms, such as [ExAllocatePoolWithTag](#), or allocating them on the stack.

The client driver must *not* use **USBD\_UrbAllocate**,

- To allocate an URB that has variable length, such as an URB for an isochronous transfer. Instead the client driver must call [USBD\\_IsochUrbAllocate](#).
- If the target operating system is Windows XP with Service Pack 2 (SP2) or an earlier version of Windows.

For more information about replacement routines, see [Allocating and Building URBS](#).

You must call [USBD\\_UrbFree](#) to release the URB allocated by **USBD\_UrbAllocate**.

## Examples

The following code example shows how to allocate, submit, and release a URB. The example submits the URB synchronously. For the implementation of the `SubmitUrbSync` function, see the example section in [How to Submit an URB](#).

C++

```
NTSTATUS CreateandSubmitURBSynchronously (
    _In_ USBD_HANDLE USBDHandle
{
    PURB    Urb = NULL;

    NTSTATUS status;

    status = USBD_UrbAllocate(USBDHandle, &Urb);

    if (!NT_SUCCESS(status))
    {
        goto CreateandSubmitURBExit;
    }

    //Format the URB for the request. Not Shown.
    status = BuildURBForBulkTransfer (Urb);

    if (!NT_SUCCESS(status))
    {
        goto CreateandSubmitURBExit;
    }
}
```

```

status = SubmitUrbSync( TargetDeviceObject,
    Urb)

if (!NT_SUCCESS(status))
{
    goto CreateandSubmitURBExit;
}

CreateandSubmitURBExit:

if (Urb)
{
    USBD_UrbFree( USBDHandle, Urb);
}

return status;

}

```

## Requirements

<b>Minimum supported client</b>	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
<b>Target Platform</b>	Desktop
<b>Header</b>	usbdlib.h
<b>Library</b>	Usbdex.lib
<b>IRQL</b>	DISPATCH_LEVEL

## See also

[Allocating and Building URBs](#)

[Sending Requests to a USB Device](#)

[USBD\\_UrbFree](#)

# USBD\_UrbFree function (usbdlib.h)

Article10/21/2021

The **USBD\_UrbFree** routine releases the **URB** that is allocated by [USBD\\_UrbAllocate](#), [USBD\\_IsochUrbAllocate](#), [USBD\\_SelectConfigUrbAllocateAndBuild](#), or [USBD\\_SelectInterfaceUrbAllocateAndBuild](#).

## Syntax

C++

```
void USBD_UrbFree(
    [in] USBD_HANDLE USBDHandle,
    [in] PURB        Urb
);
```

## Parameters

[in] **USBDHandle**

USBD handle that is retrieved by the client driver in a previous call to the [USBD\\_CreateHandle](#) routine.

[in] **Urb**

Pointer to the **URB** structure to be released.

## Return value

None

## Remarks

You must call **USBD\_UrbFree** to release the URB allocated by [USBD\\_UrbAllocate](#) after the request is complete.

Failure to call **USBD\_UrbFree** can cause a memory leak.

For a code example, see [USBD\\_UrbAllocate](#).

# Requirements

Minimum supported client	Requires WDK for Windows 8. Targets Windows Vista and later versions of the Windows operating system.
Target Platform	Desktop
Header	usbdlib.h
Library	Usbdex.lib
IRQL	<=DISPATCH_LEVEL

## See also

[Allocating and Building URBs](#)

[USBD\\_UrbAllocate](#)

# USBD\_ValidateConfigurationDescriptor function (usbdlib.h)

Article02/22/2024

The **USBD\_ValidateConfigurationDescriptor** routine validates all descriptors returned by a device in its response to a configuration descriptor request.

## Syntax

C++

```
USBD_STATUS USBD_ValidateConfigurationDescriptor(
    [in]          PUSB_CONFIGURATION_DESCRIPTOR ConfigDesc,
    [in]          ULONG             BufferLength,
    [in]          USHORT            Level,
    [out]         PUCHAR           *Offset,
    [in, optional] ULONG            Tag
);
```

## Parameters

[in] **ConfigDesc**

Pointer to a configuration descriptor that includes all interface, endpoint, vendor, and class-specific descriptors retrieved from a USB device.

[in] **BufferLength**

Size, in bytes, of the configuration descriptor being validated.

[in] **Level**

Level of validation to be performed. The following are valid values:

- 1-Basic validation of the configuration descriptor header.
- 2-Full validation of the configuration descriptor including checking for invalid endpoint addresses, interface numbers, descriptor structures, interface alternate settings, number of interfaces and **bLength** fields of all descriptors.
- 3-In addition to the validation for levels 1 and 2, level 3 validates plus validates the number of endpoints in each interface, enforces the USB specification's descriptor **bLength** sizes, and verifies that all interface numbers are in sequential order.

[out] Offset

Offset within configuration descriptor where validation failed. Only valid when a status other than USBD\_STATUS\_SUCCESS is returned.

[in, optional] Tag

Pool tag used by **USBD\_ValidateConfigurationDescriptor** when allocating memory.

## Return value

USBD\_STATUS\_SUCCESS, or appropriate USBD error code if validation failed.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Available in Windows Vista and later operating systems.
Target Platform	Universal
Header	usbdl.h (include Usbdlib.h)
Library	Usbd.lib
IRQL	PASSIVE_LEVEL

## See also

[USB device driver programming reference](#)

# usbfnattach.h header

Article 01/23/2023

This header is used by usbref. For more information, see:

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

usbfnattach.h contains the following programming interfaces:

## Callback functions

### [USBFN\\_GET\\_ATTACH\\_ACTION](#)

The filter driver's implementation that gets invoked when charger is attached to the port.

### [USBFN\\_GET\\_ATTACH\\_ACTION\\_ABORT](#)

The filter driver's implementation to abort an attach-detect operation.

### [USBFN\\_SET\\_DEVICE\\_STATE](#)

The filter driver's implementation to set the device state and operating bus speed.

## Structures

### [USBFN\\_INTERFACE\\_ATTACH](#)

Stores pointers to driver-implemented callback functions for handling attach and detach operations.

### [USBFN\\_ON\\_ATTACH](#)

Describes the detected port type and attach action.

## Enumerations

## [USBFN\\_ATTACH\\_ACTION](#)

Defines the actions that the Universal Serial Bus (USB) function stack takes when a device is attached to a USB port.

# USBFN\_ATTACH\_ACTION enumeration (usbfnattach.h)

Article 06/03/2021

Defines the actions that the Universal Serial Bus (USB) function stack takes when a device is attached to a USB port.

## Syntax

C++

```
typedef enum _USBFN_ATTACH_ACTION {
    UsbfnPortDetected,
    UsbfnPortDetectedNoCad,
    UsbfnProceedWithAttach,
    UsbfnIgnoreAttach,
    UsbfnDetectProprietaryCharger,
    UsbfnHwBasedChargerDetection
} USBFN_ATTACH_ACTION, *PUSBFN_ATTACH_ACTION;
```

## Constants

### UsbfnPortDetected

The USB function stack uses the returned port type to determine charging current and notify the Charging Aggregation Driver (CAD) of the power source change. If the detected port type is **UsbfnStandardDownstreamPort** or **UsbfnChargingDownstreamPort**, the USB function stack will attempt to connect to the host (see [USBFN\\_PORT\\_TYPE](#) for more information).

### UsbfnPortDetectedNoCad

The USB function stack does not notify the CAD of the power source change. If the detected port type is **UsbfnStandardDownstreamPort** or **UsbfnChargingDownstreamPort**, the USB function stack attempts to connect to the host (see [USBFN\\_PORT\\_TYPE](#) for more information).

### UsbfnProceedWithAttach

The USB function stack continues with the legacy software-based detection that exists in the client drivers, and issues the CAD notifications about power source notifications.

### UsbfnIgnoreAttach

The USB function stack discontinues further port detection operations and does not notify CAD of a power source update.

`UsbfnDetectProprietaryCharger`

The USB function stack calls the `UFX_PROPRIETARY_CHARGER_DETECT` event callback function implemented by the USB lower filter driver, to perform proprietary charger detection.

`UsbfnHwBasedChargerDetection`

# Requirements

Header	usbfnattach.h
--------	---------------

## See also

[USBFN\\_GET\\_ATTACH\\_ACTION](#)

# USBFN\_GET\_ATTACH\_ACTION callback function (usbfnattach.h)

Article02/22/2024

The filter driver's implementation that gets invoked when charger is attached to the port.

## Syntax

C++

```
USBFN_GET_ATTACH_ACTION UsbfnGetAttachAction;

NTSTATUS UsbfnGetAttachAction(
    [in] PVOID Context,
    [out] PUSBFN_ON_ATTACH OnAttach
)
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

[out] OnAttach

A pointer to a caller-allocated [USBFN\\_ON\\_ATTACH](#) structure that the driver populates with the type of attach and port.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support attach and detatch detection, the USB lower filter driver must publish its support. During the publishing process, the driver also registers its implementation of

this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

## Examples

```
NTSTATUS
UsbLowerFilter_GetAttachAction(
    __in PVOID Context,
    __out PUSBFN_ON_ATTACH OnAttach
)
{
    NTSTATUS Status;
    PPDCP_CONTEXT PdcpContext = NULL;
    LARGE_INTEGER Timeout;

    PAGED_CODE();

    // Get driver context
    PdcpContext = DeviceGetUsbLowerFilterContext((WDFDEVICE)Context);

    // Clear the event
    KeClearEvent(&PdcpContext->AbortAttachOperation);

    // Wait for a while
    Timeout.QuadPart = WDF_REL_TIMEOUT_IN_MS(PdcpContext-
>DetectionDelayInms);

    Status = KeWaitForSingleObject(
        &PdcpContext->AbortAttachOperation,
        Executive,
        KernelMode,
        FALSE,
        &Timeout);

    switch (Status)
    {
        case STATUS_SUCCESS:
            // The abort event was set.
            Status = STATUS_REQUEST_ABORTED;
            break;

        case STATUS_TIMEOUT:
            Status = STATUS_SUCCESS;
            break;

        default:
            break;
    }

    if (NT_SUCCESS(Status))
```

```
{  
    OnAttach->AttachAction = PdcpContext->CurrentAttachAction;  
    OnAttach->PortType = PdcpContext->CurrentPortType;  
}  
  
return Status;
```

# Requirements

[\[\] Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	usbfnattach.h
IRQL	PASSIVE_LEVEL

## See also

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

# USBFN\_GET\_ATTACH\_ACTION\_ABORT callback function (usbfnattach.h)

Article04/30/2024

The filter driver's implementation to abort an attach-detect operation.

## Syntax

C++

```
USBFN_GET_ATTACH_ACTION_ABORT UsbfnGetAttachActionAbort;

NTSTATUS UsbfnGetAttachActionAbort(
    [in] PVOID Context
)
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support attach and detach detection, the USB lower filter driver must publish its support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

## Examples

```

NTSTATUS
UsbLowerFilter_GetAttachActionAbortOperation(
    __in PVOID Context
)
{
    PPDCP_CONTEXT PdcpContext = NULL;

    PAGED_CODE();

    // Get our context
    PdcpContext = DeviceGetUsbLowerFilterContext((WDFDEVICE)Context);

    // Set our event
    (void) KeSetEvent(&PdcpContext->AbortAttachOperation,
        LOW_REALTIME_PRIORITY, FALSE);

    return STATUS_SUCCESS;
}

```

## Requirements

[] Expand table

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	usbfnattach.h
IRQL	PASSIVE_LEVEL

## See also

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

## Feedback

Was this page helpful?

👍 Yes

👎 No



# USBFN\_INTERFACE\_ATTACH structure (usbfnattach.h)

Article 02/22/2024

Stores pointers to driver-implemented callback functions for handling attach and detach operations.

## Syntax

C++

```
typedef struct _USBFN_INTERFACE_ATTACH {
    INTERFACE             InterfaceHeader;
    PFN_USBFN_GET_ATTACH_ACTION     GetAttachAction;
    PFN_USBFN_GET_ATTACH_ACTION_ABORT GetAttachActionAbortOperation;
    PFN_USBFN_SET_DEVICE_STATE      SetDeviceState;
} USBFN_INTERFACE_ATTACH, *PUSBFN_INTERFACE_ATTACH;
```

## Members

`InterfaceHeader`

The interface version number.

`GetAttachAction`

A pointer to the driver's implementation of the [USBFN\\_GET\\_ATTACH\\_ACTION](#) callback function.

`GetAttachActionAbortOperation`

A pointer to the driver's implementation of the [USBFN\\_GET\\_ATTACH\\_ACTION\\_ABORT](#) callback function.

`SetDeviceState`

A pointer to the driver's implementation of the [USBFN\\_SET\\_DEVICE\\_STATE](#) callback function.

## Requirements

[+] Expand table

Requirement	Value
Header	usbfnattach.h

## See also

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

# USBFN\_ON\_ATTACH structure (usbfnattach.h)

Article 02/22/2024

Describes the detected port type and attach action.

## Syntax

C++

```
typedef struct _USBFN_ON_ATTACH {
    USBFN_PORT_TYPE      PortType;
    USBFN_ATTACH_ACTION AttachAction;
} USBFN_ON_ATTACH, *PUSBFN_ON_ATTACH;
```

## Members

PortType

Detected port type defined by one of the [USBFN\\_PORT\\_TYPE](#)-typed values.

AttachAction

The operation that must be performed depending on the port type. This value is defined in the [USBFN\\_ATTACH\\_ACTION](#) enumeration.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbfnattach.h

## See also

[USBFN\\_GET\\_ATTACH\\_ACTION](#)

# USBFN\_SET\_DEVICE\_STATE callback function (usbfnattach.h)

Article02/22/2024

The filter driver's implementation to set the device state and operating bus speed.

## Syntax

```
C++  
  
USBFN_SET_DEVICE_STATE UsbfnSetDeviceState;  
  
NTSTATUS UsbfnSetDeviceState(  
    [in] PVOID Context,  
    [in] USBFN_DEVICE_STATE DeviceState,  
    [in] USBFN_BUS_SPEED BusSpeed  
)  
{...}
```

## Parameters

[in] Context

A pointer to a driver-defined context.

[in] DeviceState

A [USBFN\\_DEVICE\\_STATE](#)-typed flag that indicates the state of the device.

[in] BusSpeed

A [USBFN\\_BUS\\_SPEED](#)-typed flag that indicates the bus speed.

## Return value

If the operation is successful, the callback function must return STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it must return a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

To support attach and detatch detection, the USB lower filter driver must publish its support. During the publishing process, the driver also registers its implementation of this callback function. For more information, see [USB filter driver for supporting proprietary chargers](#).

The lower filter driver might implement a *USBFN\_SET\_DEVICE\_STATE* even callback function if it requires notification of device state changes to properly configure charging when attached to a host, or in lab scenarios where charging via USB must be disabled.

## Examples

```
NTSTATUS
UsbLowerFilter_SetDeviceState(
    _In_ PVOID Context,
    _In_ USBFN_DEVICE_STATE DeviceState,
    _In_ USBFN_BUS_SPEED BusSpeed
)
{
    PPDPC_CONTEXT PdcpContext = NULL;

    PAGED_CODE();

    // Get our context
    PdcpContext = DeviceGetUsbLowerFilterContext((WDFDEVICE)Context);

    PdcpContext->CurrentDeviceState = DeviceState;
    PdcpContext->BusSpeed = BusSpeed;

    return STATUS_SUCCESS;
}
```

## Requirements

[ ] [Expand table](#)

Requirement	Value
Target Platform	Windows
Minimum KMDF version	1.0
Minimum UMDF version	2.0
Header	usbfnattach.h

Requirement	Value
IRQL	PASSIVE_LEVEL

## See also

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

# usbfnbase.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usbfnbase.h contains the following programming interfaces:

## Structures

### [ALTERNATE\\_INTERFACE](#)

The ALTERNATE\_INTERFACE structure provides information about alternate settings for a Universal Serial Bus (USB) interface.

### [USBFN\\_BUS\\_CONFIGURATION\\_INFO](#)

Configuration packet that stores information about an available USB configuration.

### [USBFN\\_CLASS\\_INFORMATION\\_PACKET](#)

Describes device interface class information associated with a USB interface. This structure can only hold information about a single function interface.

### [USBFN\\_CLASS\\_INFORMATION\\_PACKET\\_EX](#)

Describes device interface class information associated with a USB interface. This structure can be used to describe single and multi-interface functions.

### [USBFN\\_CLASS\\_INTERFACE](#)

Describes an interface and its endpoints.

### [USBFN\\_CLASS\\_INTERFACE\\_EX](#)

Learn how USBFN\_CLASS\_INTERFACE\_EX describes an interface and its endpoints.

### [USBFN\\_INTERFACE\\_INFO](#)

Learn how USBFN\_INTERFACE\_INFO describes an interface and its endpoints.

### [USBFN\\_NOTIFICATION](#)

Describes information about a Universal Serial Bus (USB) event notification that was received by using IOCTL\_INTERNAL\_USBFN\_BUS\_EVENT\_NOTIFICATION.

#### [USBFN\\_PIPE\\_INFORMATION](#)

Describes attributes of a pipe associated with an endpoint on a specific interface.

#### [USBFN\\_USB\\_STRING](#)

Describes a USB string descriptor and the associated string index.

## Enumerations

#### [USBFN\\_BUS\\_SPEED](#)

The USBFN\_BUS\_SPEED enumeration defines possible bus speeds.

#### [USBFN\\_DEVICE\\_STATE](#)

Defines the Universal Serial Bus (USB) device states for the device/controller. These states correspond to the USB device states as defined in section 9.1 of the USB 2.0 Specification.

#### [USBFN\\_DIRECTION](#)

Defines the USB data transfer direction types.

#### [USBFN\\_EVENT](#)

Defines notifications sent to class drivers.

#### [USBFN\\_PORT\\_TYPE](#)

Defines the possible port types that can be returned by the client driver during port detection.

# ALTERNATE\_INTERFACE structure (usbfbase.h)

Article02/22/2024

The **ALTERNATE\_INTERFACE** structure provides information about alternate settings for a Universal Serial Bus (USB) interface.

## Syntax

C++

```
typedef struct _ALTERNATE_INTERFACE {
    USHORT InterfaceNumber;
    USHORT AlternateInterfaceNumber;
} ALTERNATE_INTERFACE, *PALTERNATE_INTERFACE;
```

## Members

InterfaceNumber

The index number for the USB interface setting.

AlternateInterfaceNumber

The index number for the alternate USB interface setting.

## Requirements

[+] Expand table

Requirement	Value
Header	usbfbase.h (include Usbfbase.h)

## See also

[USBFN\\_NOTIFICATION](#)

# USBFN\_BUS\_CONFIGURATION\_INFO structure (usbfnbase.h)

Article02/22/2024

Configuration packet that stores information about an available USB configuration.

## Syntax

C++

```
typedef struct _USBFN_BUS_CONFIGURATION_INFO {
    WCHAR ConfigurationName[MAX_CONFIGURATION_NAME_LENGTH];
    BOOLEAN IsCurrent;
    BOOLEAN IsActive;
} USBFN_BUS_CONFIGURATION_INFO, *PUSBFN_BUS_CONFIGURATION_INFO;
```

## Members

ConfigurationName[MAX\_CONFIGURATION\_NAME\_LENGTH]

A NULL-terminated string that indicates the name of a configuration.

IsCurrent

Indicates whether this configuration is the current configuration.

IsActive

Indicates whether the configuration is active. This is a read-only information that is returned by USB function class extension (UFX) and is ignored in requests sent to UFX.

## Requirements

[ ] Expand table

Requirement	Value
Header	usbfnbase.h

# USBFN\_BUS\_SPEED enumeration (usbfnbase.h)

Article 02/22/2024

The **USBFN\_BUS\_SPEED** enumeration defines possible bus speeds.

## Syntax

C++

```
typedef enum _USBFN_BUS_SPEED {
    UsbfnBusSpeedLow,
    UsbfnBusSpeedFull,
    UsbfnBusSpeedHigh,
    UsbfnBusSpeedSuper,
    UsbfnBusSpeedMaximum
} USBFN_BUS_SPEED, *PUSBFN_BUS_SPEED;
```

## Constants

[+] Expand table

**UsbfnBusSpeedLow**

A low bandwidth bus speed of 1.5 Mbit per second.

**UsbfnBusSpeedFull**

A full bandwidth bus speed of 12 MBit per second.

**UsbfnBusSpeedHigh**

A high bus speed of 480 Mbit per second.

**UsbfnBusSpeedSuper**

A SuperSpeed mode bus speed of 5 Gbit per second.

**UsbfnBusSpeedMaximum**

The maximum value of the enumeration.

## Requirements

 Expand table

Requirement	Value
Header	usbfnbase.h

# USBFN\_CLASS\_INFORMATION\_PACKET structure (usbfnbase.h)

Article02/22/2024

Describes device interface class information associated with a USB interface. This structure can only hold information about a single function interface.

## Syntax

C++

```
typedef struct _USBFN_CLASS_INFORMATION_PACKET {
    USBFN_CLASS_INTERFACE FullSpeedClassInterface;
    USBFN_CLASS_INTERFACE HighSpeedClassInterface;
    WCHAR                 InterfaceName[MAX_INTERFACE_NAME_LENGTH];
    WCHAR                 InterfaceGuid[MAX_INTERFACE_GUID_LENGTH];
    BOOLEAN                HasInterfaceGuid;
    USBFN_CLASS_INTERFACE SuperSpeedClassInterface;
} USBFN_CLASS_INFORMATION_PACKET, *PUSBFN_CLASS_INFORMATION_PACKET;
```

## Members

`FullSpeedClassInterface`

A [USBFN\\_CLASS\\_INTERFACE](#) structure that describes an interface for full speed device.

`HighSpeedClassInterface`

A [USBFN\\_CLASS\\_INTERFACE](#) structure that describes an interface for high speed device.

`InterfaceName[MAX_INTERFACE_NAME_LENGTH]`

A string that contains the interface name.

`InterfaceGuid[MAX_INTERFACE_GUID_LENGTH]`

A string from which the driver can derive the device interface GUID.

`HasInterfaceGuid`

Determines whether the driver has published a device interface is GUID.

`SuperSpeedClassInterface`

A [USBFN\\_CLASS\\_INTERFACE](#) structure that describes an interface for SuperSpeed device.

## Requirements

 [Expand table](#)

Requirement	Value
Header	usbfnbase.h

## See also

- [USBFN\\_CLASS\\_INTERFACE](#)
- [WdfDeviceCreateSymbolicLink](#)
- [WdfDeviceSetDeviceInterfaceState](#)

# USBFN\_CLASS\_INFORMATION\_PACKET\_EX structure (usbfnbase.h)

Article02/22/2024

Describes device interface class information associated with a USB interface. This structure can be used to describe single and multi-interface functions.

## Syntax

C++

```
typedef struct _USBFN_CLASS_INFORMATION_PACKET_EX {
    USBFN_CLASS_INTERFACE_EX FullSpeedClassInterfaceEx;
    USBFN_CLASS_INTERFACE_EX HighSpeedClassInterfaceEx;
    USBFN_CLASS_INTERFACE_EX SuperSpeedClassInterfaceEx;
    WCHAR                 InterfaceName[MAX_INTERFACE_NAME_LENGTH];
    WCHAR                 InterfaceGuid[MAX_INTERFACE_GUID_LENGTH];
    BOOLEAN               HasInterfaceGuid;
} USBFN_CLASS_INFORMATION_PACKET_EX, *PUSBFN_CLASS_INFORMATION_PACKET_EX;
```

## Members

`FullSpeedClassInterfaceEx`

A [USBFN\\_CLASS\\_INTERFACE\\_EX](#) structure that describes an interface for full speed device.

`HighSpeedClassInterfaceEx`

A structure that describes an interface for high speed device.

`SuperSpeedClassInterfaceEx`

A [USBFN\\_CLASS\\_INTERFACE\\_EX](#) structure that describes an interface for SuperSpeed device.

`InterfaceName[MAX_INTERFACE_NAME_LENGTH]`

A string that contains the interface name.

`InterfaceGuid[MAX_INTERFACE_GUID_LENGTH]`

A string from which the driver can derive the device interface GUID.

## HasInterfaceGuid

Determines whether the driver has published a device interface is GUID.

# Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbfnbase.h

## See also

- [USBFN\\_CLASS\\_INTERFACE\\_EX](#)
- [WdfDeviceCreateSymbolicLink](#)
- [WdfDeviceSetDeviceInterfaceStateEx](#)

# USBFN\_CLASS\_INTERFACE structure (usbfnbase.h)

Article 02/22/2024

Describes an interface and its endpoints.

## Syntax

C++

```
typedef struct _USBFN_CLASS_INTERFACE {
    UINT8             InterfaceNumber;
    UINT8             PipeCount;
    USBFN_PIPE_INFORMATION PipeArr[MAX_NUM_USBFN_PIPES];
} USBFN_CLASS_INTERFACE, *PUSBFN_CLASS_INTERFACE;
```

## Members

InterfaceNumber

The index number of the interface.

PipeCount

The number of endpoints contained in the interface.

PipeArr[MAX\_NUM\_USBFN\_PIPES]

An array of [USBFN\\_PIPE\\_INFORMATION](#) structures that describes the endpoints in the interface.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbfnbase.h

## See also

[USBFN\\_PIPE\\_INFORMATION](#)

# USBFN\_CLASS\_INTERFACE\_EX structure (usbfnbase.h)

Article02/22/2024

Describes an interface and its endpoints.

## Syntax

C++

```
typedef struct _USBFN_CLASS_INTERFACE_EX {
    UINT8             BaseInterfaceNumber;
    UINT8             InterfaceCount;
    UINT8             PipeCount;
    USBFN_PIPE_INFORMATION PipeArr[MAX_NUM_USBFN_PIPES];
} USBFN_CLASS_INTERFACE_EX, *PUSBFN_CLASS_INTERFACE_EX;
```

## Members

BaseInterfaceNumber

The index number of the interface.

InterfaceCount

The number of USB interfaces contained in the selected function.

PipeCount

The number of endpoints contained in the interface.

PipeArr[MAX\_NUM\_USBFN\_PIPES]

An array of [USBFN\\_PIPE\\_INFORMATION](#) structures that describes the endpoints in the interface.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbfnbase.h

# USBFN\_DEVICE\_STATE enumeration (usbfnbase.h)

Article 02/22/2024

Defines the Universal Serial Bus (USB) device states for the device/controller. These states correspond to the USB device states as defined in section 9.1 of the USB 2.0 Specification.

## Syntax

C++

```
typedef enum _USBFN_DEVICE_STATE {
    UsbfnDeviceStateMinimum,
    UsbfnDeviceStateAttached,
    UsbfnDeviceStateDefault,
    UsbfnDeviceStateDetached,
    UsbfnDeviceStateAddressed,
    UsbfnDeviceStateConfigured,
    UsbfnDeviceStateSuspended,
    UsbfnDeviceStateStateMaximum
} USBFN_DEVICE_STATE, *PUSBFN_DEVICE_STATE;
```

## Constants

[+] Expand table

<b>UsbfnDeviceStateMinimum</b> The minimum value of the enumeration.
<b>UsbfnDeviceStateAttached</b> Device is attached to an upstream port.
<b>UsbfnDeviceStateDefault</b> Device is attached and connected to an upstream port but has not been reset.
<b>UsbfnDeviceStateDetached</b> Device is not attached to an upstream port.
<b>UsbfnDeviceStateAddressed</b> Device has been assigned a non-default USB address by the host.

**UsbfnDeviceStateConfigured**

Device has been configured by the host.

**UsbfnDeviceStateSuspended**

Device has been suspended.

**UsbfnDeviceStateStateMaximum**

The maximum value of the enumeration.

# Requirements

[\[\] Expand table](#)

Requirement	Value
Header	usbfnbase.h

# USBFN\_DIRECTION enumeration (usbfnbase.h)

Article 02/22/2024

Defines the USB data transfer direction types.

## Syntax

C++

```
typedef enum _USBFN_DIRECTION {
    UsbfnDirectionMinimum,
    UsbfnDirectionIn,
    UsbfnDirectionOut,
    UsbfnDirectionTx,
    UsbfnDirectionRx,
    UsbfnDirectionMaximum
} USBFN_DIRECTION, *PUSBFN_DIRECTION;
```

## Constants

[\[+\] Expand table](#)

<b>UsbfnDirectionMinimum</b> The minimum value in this enumeration.
<b>UsbfnDirectionIn</b> The transfer is to the host from an endpoint.
<b>UsbfnDirectionOut</b> The transfer is from the host to the endpoint.
<b>UsbfnDirectionTx</b> The bus transfer to the host from the device.
<b>UsbfnDirectionRx</b> The bus transfer is from the host to the device.
<b>UsbfnDirectionMaximum</b> The maximum value in this enumeration.

# Requirements

 [Expand table](#)

Requirement	Value
Header	usbfnbase.h

# USBFN\_EVENT enumeration (usbfnbase.h)

Article 02/22/2024

Defines notifications sent to class drivers.

## Syntax

C++

```
typedef enum _USBFN_EVENT {
    UsbfnEventMinimum,
    UsbfnEventAttach,
    UsbfnEventReset,
    UsbfnEventDetach,
    UsbfnEventSuspend,
    UsbfnEventResume,
    UsbfnEventSetupPacket,
    UsbfnEventConfigured,
    UsbfnEventUnConfigured,
    UsbfnEventPortType,
    UsbfnEventBusTearDown,
    UsbfnEventSetInterface,
    UsbfnEventMaximum
} USBFN_EVENT, *PUSBFN_EVENT;
```

## Constants

[ ] Expand table

`UsbfnEventMinimum`

The minimum value in this enumeration.

`UsbfnEventAttach`

VBUS is powered. No action is required.

`UsbfnEventReset`

USBFN has completed a USB Reset. If previously configured, class drivers should reset their state. Transfer requests will be cancelled.

`UsbfnEventDetach`

VBUS is no longer powered.

If previously configured, class drivers should reset their state. Transfer requests will be cancelled. The **BusSpeed** field of the notification is set appropriately.

#### UsbfnEventSuspend

There have been no SOF packets on the bus for 3ms. If a class driver wants to issue a remote wake up, the driver must use [IOCTL\\_INTERNAL\\_USBFN\\_SIGNAL\\_REMOTE\\_WAKEUP](#) or [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN](#).

#### UsbfnEventResume

USBFN has resumed from suspend to the previous state.

#### UsbfnEventSetupPacket

USBFN has received a setup packet with **bmRequestType.Type** set to BMREQUEST\_CLASS and **bmRequestType.Recipient** set to BMREQUEST\_TO\_INTERFACE. USBFN forwarded the setup packet to the class driver specified in **wIndex.LowByte**.

The setup packet is available in the **SetupPacket** field of the event. If the control transfer does not require a data stage, class drivers should respond with

[IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_OUT](#).

If a data stage is required, class drivers should respond with one or more [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN](#) or [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_OUT](#), followed by [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_IN](#) or [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_OUT](#) in the opposite direction.

#### UsbfnEventConfigured

USBFN has received a SET\_CONFIGURATION setup packet. Transfer requests from class drivers are now permitted.

The **ConfigurationValue** of the notification is set to **wValue.W**.

#### UsbfnEventUnConfigured

USBFN has received a SET\_CONFIGURATION setup packet with **wValue.W** set to 0. If previously configured, class drivers should reset their state. Transfer requests will be cancelled.

#### UsbfnEventPortType

Deprecated.

#### UsbfnEventBusTearDown

Deprecated.

#### UsbfnEventSetInterface

USBFN has received a SET\_INTERFACE setup packet. On receiving this

notification the class driver should query for the new endpoint set for the interface.

**UsbfnEventMaximum**

The minimum value in this enumeration.

# Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	usbfnbase.h

# USBFN\_INTERFACE\_INFO structure (usbfnbase.h)

Article 02/22/2024

Describes an interface and its endpoints.

## Syntax

C++

```
typedef struct _USBFN_INTERFACE_INFO {
    UINT8           InterfaceNumber;
    USBFN_BUS_SPEED Speed;
    USHORT          Size;
    UCHAR           InterfaceDescriptorSet[1];
} USBFN_INTERFACE_INFO, *PUSBFN_INTERFACE_INFO;
```

## Members

`InterfaceNumber`

The index number of the interface.

`Speed`

The operating bus speed indicated by `USBFN_BUS_SPEED`-typed flags.

`Size`

Specifies the total length, in bytes, of all data for the interface.

`InterfaceDescriptorSet[1]`

Pointer to the first element in the array of that contains the interface descriptor set.

## Requirements

[+] Expand table

Requirement	Value
Header	usbfnbase.h

## See also

[USBFN\\_BUS\\_SPEED](#)

# USBFN\_NOTIFICATION structure (usbfnbase.h)

Article 02/22/2024

Describes information about a Universal Serial Bus (USB) event notification that was received by using [IOCTL\\_INTERNAL\\_USBFN\\_BUS\\_EVENT\\_NOTIFICATION](#).

## Syntax

C++

```
typedef struct _USBFN_NOTIFICATION {
    USBFN_EVENT Event;
    union {
        USBFN_BUS_SPEED             BusSpeed;
        USB_DEFAULT_PIPE_SETUP_PACKET SetupPacket;
        USHORT                      ConfigurationValue;
        USBFN_PORT_TYPE              PortType;
        ALTERNATE_INTERFACE          AlternateInterface;
    } u;
} USBFN_NOTIFICATION, *PUSBFN_NOTIFICATION;
```

## Members

Event

Bus notification indicated by a [USBFN\\_EVENT](#)-typed flag.

u

u.BusSpeed

The operating bus speed indicated by [USBFN\\_BUS\\_SPEED](#)-typed flags.

u.SetupPacket

Describes a setup packet in a [USB\\_DEFAULT\\_PIPE\\_SETUP\\_PACKET](#) structure for a control transfer to or from the default endpoint as indicated by a [USB\\_DEFAULT\\_PIPE\\_SETUP\\_PACKET](#)-typed flag.

u.ConfigurationValue

The bConfigurationValue field of a USB configuration descriptor.

### `u.PortType`

Possible port types supported by a function controller indicated by a [USBFN\\_PORT\\_TYPE](#)-typed flag.

### `u.AlternateInterface`

Alternate setting of the interface indicated by [ALTERNATE\\_INTERFACE](#).

## Requirements

 Expand table

Requirement	Value
Header	usbfnbase.h

# USBFN\_PIPE\_INFORMATION structure (usbfnbase.h)

Article 02/22/2024

Describes attributes of a pipe associated with an endpoint on a specific interface.

## Syntax

C++

```
typedef struct _USBFN_PIPE_INFORMATION {
    USB_ENDPOINT_DESCRIPTOR EpDesc;
    USBFNPIPEID PipeId;
} USBFN_PIPE_INFORMATION, *PUSBFN_PIPE_INFORMATION;
```

## Members

EpDesc

Describes the endpoint descriptor in a [USB\\_ENDPOINT\\_DESCRIPTOR](#) structure.

PipeId

The pipe identifier (ID).

## Requirements

[Expand table](#)

Requirement	Value
Header	usbfnbase.h

## See also

[USB\\_ENDPOINT\\_DESCRIPTOR](#)

# USBFN\_PORT\_TYPE enumeration (usbfnbase.h)

Article 06/03/2021

Defines the possible port types that can be returned by the client driver during port detection.

## Syntax

C++

```
typedef enum _USBFN_PORT_TYPE {
    UsbfnUnknownPort,
    UsbfnStandardDownstreamPort,
    UsbfnChargingDownstreamPort,
    UsbfnDedicatedChargingPort,
    UsbfnInvalidDedicatedChargingPort,
    UsbfnProprietaryDedicatedChargingPort,
    UsbfnPortTypeMaximum
} USBFN_PORT_TYPE, *PUSBFN_PORT_TYPE;
```

## Constants

**UsbfnUnknownPort**

Port detection was unable to determine the port type.

**UsbfnStandardDownstreamPort**

The upstream port has been detected as a standard downstream port (SDP) (as defined in the Battery Charging Specification, revision 1.2).

**UsbfnChargingDownstreamPort**

The upstream port has been detected as a charging downstream port (CDP), as defined in the Battery Charging Specification, revision 1.2.

**UsbfnDedicatedChargingPort**

The upstream port has been detected as a dedicated charging port (DCP) (as defined in the Battery Charging Specification, revision 1.2).

**UsbfnInvalidDedicatedChargingPort**

The upstream port has been detected as a dedicated charging port that does not comply with the Battery Charging Specification, revision 1.2.

`UsbfnProprietaryDedicatedChargingPort`

A proprietary charger was attached.

`UsbfnPortTypeMaximum`

The maximum value of the enumeration.

# Requirements

Header

`usbfnbase.h`

## See also

[USBFN\\_GET\\_ATTACH\\_ACTION](#)

# USBFN\_USB\_STRING structure (usbfnbase.h)

Article02/22/2024

Describes a USB string descriptor and the associated string index.

## Syntax

C++

```
typedef struct _USBFN_USB_STRING {
    UINT8 StringIndex;
    WCHAR UsbString[MAX_USB_STRING_LENGTH];
} USBFN_USB_STRING, *PUSBFN_USB_STRING;
```

## Members

`StringIndex`

The string index.

`UsbString[MAX_USB_STRING_LENGTH]`

Pointer to the string.

## Requirements

 Expand table

Requirement	Value
Header	usbfnbase.h

## See also

[IOCTL\\_INTERNAL\\_USBFN\\_REGISTER\\_USB\\_STRING](#)

# usbfnioctl.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usbfnioctl.h contains the following programming interfaces:

## IOCTLS

### [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#)

The USB class driver sends this request to activate the bus so that the driver can prepare to process bus events and handle traffic.

### [IOCTL\\_INTERNAL\\_USBFN\\_BUS\\_EVENT\\_NOTIFICATION](#)

The USB class driver sends this request to prepare for notifications received from the USB function class extension (UFx) in response to an event on the bus, such as a change in the port type or a receipt of a non-standard setup packet.

### [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_IN](#)

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the IN direction.

### [IOCTL\\_INTERNAL\\_USBFN\\_CONTROL\\_STATUS\\_HANDSHAKE\\_OUT](#)

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the OUT direction.

### [IOCTL\\_INTERNAL\\_USBFN\\_DEACTIVATE\\_USB\\_BUS](#)

Do not use.

### [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_CLASS\\_INFO](#)

The class driver sends this request IO control code to retrieve information about the available pipes for a device, as configured in the registry.

### [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_INTERFACE\\_DESCRIPTOR\\_SET](#)

The class driver sends this request to get the entire USB interface descriptor set for a function on the device.

## [IOCTL\\_INTERNAL\\_USBFN\\_GET\\_PIPE\\_STATE](#)

The class driver sends this request to get the stall state of the specified pipe.

## [IOCTL\\_INTERNAL\\_USBFN\\_REGISTER\\_USB\\_STRING](#)

The class driver sends this request to register a USB string descriptor.

## [IOCTL\\_INTERNAL\\_USBFN\\_RESERVED](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_RESERVED) article.

## [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_PIPE\\_STATE](#)

The class driver sends this request to set the stall state of the specified USB pipe.

## [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_EXIT\\_LPM](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_EXIT\_LPM) article.

## [IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_STATE](#)

Do not use this (IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_STATE) article.

## [IOCTL\\_INTERNAL\\_USBFN\\_SIGNAL\\_REMOTE\\_WAKEUP](#)

The class driver sends this request to get remote wake-up notifications from endpoints.

## [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN](#)

The class driver sends this request to initiate a data transfer to the host on the specified pipe.

## [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_IN\\_APPEND\\_ZERO\\_PKT](#)

The class driver sends this request to initiate an IN transfer to the specified pipe and appends a zero-length packet to indicate the end of the transfer.

## [IOCTL\\_INTERNAL\\_USBFN\\_TRANSFER\\_OUT](#)

The class driver sends this request to initiate a data transfer from the host on the specified pipe.

# Structures

## [USBFN\\_POWER\\_FILTER\\_STATE](#)

Reserved. Do not use.

# **IOCTL\_INTERNAL\_USBFN\_ACTIVATE\_US B\_BUS IOCTL (usbfnioclt.h)**

Article02/22/2024

The USB class driver sends this request to activate the bus so that the driver can prepare to process bus events and handle traffic.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

NULL.

## **Input buffer length**

None.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

All class drivers must send this IOCTL request before the device attempts to connect with the host.

# Requirements

 Expand table

Requirement	Value
Header	usbfnioclt.h (include Usbfnioclt.h)

# **IOCTL\_INTERNAL\_USBFN\_BUS\_EVENT\_NOTIFICATION IOCTL (usbfniioctl.h)**

Article06/16/2023

The USB class driver sends this request to prepare for notifications received from the USB function class extension (UFX) in response to an event on the bus, such as a change in the port type or a receipt of a non-standard setup packet.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

NULL.

## **Input buffer length**

None.

## **Output buffer**

A pointer to a caller-allocated [USBFN\\_NOTIFICATION](#) structure that UFX populates with the type of bus event and data associated with that event.

## **Output buffer length**

The size of a [USBFN\\_NOTIFICATION](#) structure.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

UFX completes this request in response to an event on the bus. It is recommended that class drivers send multiple requests at a time to make sure that critical notifications are not missed.

## Requirements

Header	usbfnioc.h
--------	------------

## See also

[USBFN\\_EVENT](#)

[USBFN\\_NOTIFICATION](#)

# **IOCTL\_INTERNAL\_USBFN\_CONTROL\_ST ATUS\_HANDSHAKE\_IN IOCTL (usbfnioctl.h)**

Article02/22/2024

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the IN direction.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A **USBFNPIPEID** type value that indicates the pipe ID. The pipe ID of the default control endpoint is 0.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

NULL.

## **Output buffer length**

NULL.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Header	usbfioctl.h

# **IOCTL\_INTERNAL\_USBFN\_CONTROL\_STATUS\_HANDSHAKE\_OUT IOCTL (usbfnioctl.h)**

Article02/22/2024

The class driver sends this request to send a zero-length control status handshake on endpoint 0 in the OUT direction.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A **USBFNPIPEID** type value that indicates the pipe ID. The pipe ID of the default control endpoint is 0.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns **STATUS\_SUCCESS**, or another status value for which **NT\_SUCCESS(status)** equals TRUE. Otherwise it returns a status value for which **NT\_SUCCESS(status)** equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Header	usbfioctl.h

# **IOCTL\_INTERNAL\_USBFN\_DEACTIVATE\_USB\_BUS IOCTL (usbfnioclt.h)**

Article02/22/2024

Do not use.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

NULL.

## **Input buffer length**

None.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Requirements**

 Expand table

Requirement	Value
Header	usbfniocli.h (include Usbfniocli.h)

# **IOCTL\_INTERNAL\_USBFN\_GET\_CLASS\_INFORMATION IOCTL (usbfniioctl.h)**

Article06/16/2023

The class driver sends this request IO control code to retrieve information about the available pipes for a device, as configured in the registry.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

NULL.

## **Input buffer length**

None.

## **Output buffer**

A pointer to a buffer that contains a [USBFN\\_CLASS\\_INFORMATION\\_PACKET](#) structure.

Upon completion, UFX populates the structure with the name, the device interface GUID, and details of the interface when operating at a particular bus speed.

## **Output buffer length**

The size of a [USBFN\\_CLASS\\_INFORMATION\\_PACKET](#) structure.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

The class driver should send this IOCTL request during initialization so that it can enumerate the endpoints and attributes.

## Requirements

<b>Header</b>	usbfioctl.h

# IOCTL\_INTERNAL\_USBFN\_GET\_INTERFACE\_DESCRIPTOR\_SET IOCTL (usbfnioctl.h)

Article06/16/2023

The class driver sends this request to get the entire USB interface descriptor set for a function on the device.

**Note** Do not use this request to retrieve the interface descriptor set for the entire device.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

A pointer to a buffer that contains a [USBFN\\_INTERFACE\\_INFO](#) structure.

## Input buffer length

The length of the input buffer must be at least `sizeof(USBFN_INTERFACE_INFO)`.

## Output buffer

A pointer to a buffer that contains a [USBFN\\_INTERFACE\\_INFO](#) structure. USB function class extension (UFX) populates the structure with the entire interface descriptor set including its endpoint descriptors.

## Output buffer length

The length of the output buffer must be at least `sizeof(USBFN_INTERFACE_INFO)`.

## Status block

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## Remarks

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

The length of the entire interface descriptor is variable. The class driver might need to send this IOCTL request twice to get the entire descriptor set.

If the length of the entire descriptor set is greater than the specified output buffer length, UFX sets the **Size** member of [USBFN\\_INTERFACE\\_INFO](#) to the actual buffer length and fails the request with STATUS\_BUFFER\_TOO\_SMALL. The driver must then allocate an output buffer of length specified by **Size** and resend the request.

## Requirements

Header	usbfnioclt.h
--------	--------------

## See also

[IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#)

[USBFN\\_INTERFACE\\_INFO](#)

# **IOCTL\_INTERNAL\_USBFN\_GET\_PIPE\_STA TE IOCTL (usbfniioctl.h)**

Article02/22/2024

The class driver sends this request to get the stall state of the specified pipe.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a **USBFNPIPEID** type that specifies the pipe ID.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

A pointer to **BOOLEAN** value that is set by USB Function Class Extension (UFX) to indicate whether or not the specified pipe is stalled. TRUE, indicates the pipe is in stall state; FALSE indicates the pipe is in clear state.

## **Output buffer length**

The size of a **BOOLEAN**.

## **Status block**

UFX completes the request with **STATUS\_SUCCESS**.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

When stalled, the pipe sends STALL transaction packets to the host. See the Universal Serial Bus (USB) specification for more information.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

## Requirements

[+] Expand table

Requirement	Value
Header	usbfioctl.h

# **IOCTL\_INTERNAL\_USBFN\_REGISTER\_USB\_STRING IOCTL (usbfnioclt.h)**

Article02/22/2024

The class driver sends this request to register a USB string descriptor.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a buffer that contains a [USBFN\\_USB\\_STRING](#) structure with the USB string descriptor.

## **Input buffer length**

The length of the input buffer must be at least `sizeof(USBFN_USB_STRING)`.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

# Requirements

 Expand table

Requirement	Value
Header	usbfioctl.h

## See also

[IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#)

[USBFN\\_USB\\_STRING](#)

# **IOCTL\_INTERNAL\_USBFN\_RESERVED**

## **IOCTL (usbfnioclt.h)**

Article02/22/2024

Do not use.

### **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### **Input buffer**

None.

### **Input buffer length**

None.

### **Output buffer**

None.

### **Output buffer length**

None.

### **Status block**

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## **Requirements**

 Expand table

<b>Requirement</b>	<b>Value</b>
Header	usbfniocli.h (include Usbfniocli.h)

# **IOCTL\_INTERNAL\_USBFN\_SET\_PIPE\_STAT**

## **E IOCTL (usbfnioclt.h)**

Article02/22/2024

The class driver sends this request to set the stall state of the specified USB pipe.

### **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### **Input buffer**

A pointer to a **USBFNPIPEID** type that specifies the pipe ID.

### **Input buffer length**

The size of a **USBFNPIPEID** type.

### **Output buffer**

A pointer to **BOOLEAN** value that specifies the stall state to set. If TRUE, USB Function Class Extension (UFX) sets the pipe to stall state; FALSE sets to clear state.

### **Output buffer length**

The size of a **BOOLEAN**.

### **Status block**

UFX completes the request with **STATUS\_SUCCESS**.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

# Requirements

 Expand table

Requirement	Value
Header	usbfioctl.h

# **IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_EXIT\_LPM IOCTL (usbfniioctl.h)**

Article02/22/2024

Do not use.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

None.

## **Input buffer length**

None.

## **Output buffer**

None.

## **Output buffer length**

None.

## **Status block**

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

Upon receiving this request, the filter driver communicates with the hardware and brings the controller out of LPM.

## Requirements

 Expand table

Requirement	Value
Header	usbfnioctl.h (include Usbfnioctl.h)

## See also

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

# **IOCTL\_INTERNAL\_USBFN\_SET\_POWER\_FILTER\_STATE IOCTL (usbfniioctl.h)**

Article02/22/2024

Do not use. ucxc

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

The input buffer contains a [USBFN\\_POWER\\_FILTER\\_STATE](#) structure that specifies the device state.

## **Input buffer length**

The size of a [USBFN\\_POWER\\_FILTER\\_STATE](#) structure.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## **Requirements**

[+] [Expand table](#)

<b>Requirement</b>	<b>Value</b>
Header	usbfniocli.h (include Usbfniocli.h)

# **IOCTL\_INTERNAL\_USBFN\_SIGNAL\_REMOVE\_WAKEUP IOCTL (usbfniioctl.h)**

Article06/16/2023

The class driver sends this request to get remote wake-up notifications from endpoints.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

NULL.

## **Input buffer length**

None.

## **Output buffer**

NULL.

## **Output buffer length**

None.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

The USB function class extension (UFX) determines the endpoints that are remote wake-up capable and registers for remote wake notifications.

## Requirements

Header	usbfioctl.h
--------	-------------

# **IOCTL\_INTERNAL\_USBFN\_TRANSFER\_IN IOCTL (usbfnioclt.h)**

Article06/16/2023

The class driver sends this request to initiate a data transfer to the host on the specified pipe.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a **USBFNPIPEID** type that specifies the pipe ID.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

The output buffer points to a buffer containing the data to be sent. The IN direction is from the host perspective representing an outbound transfer from the device to the host.

## **Output buffer length**

The length of the data to be sent.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns STATUS\_SUCCESS, or another status value for which NT\_SUCCESS(status) equals TRUE. Otherwise it returns a status value for which NT\_SUCCESS(status) equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

## Requirements

Header	usbfnioclt.h
--------	--------------

# **IOCTL\_INTERNAL\_USBFN\_TRANSFER\_IN \_APPEND\_ZERO\_PKT IOCTL (usbfnioctl.h)**

Article02/22/2024

The class driver sends this request to initiate an IN transfer to the specified pipe and appends a zero-length packet to indicate the end of the transfer.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a **USBFNPIPEID** type that specifies the pipe ID.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

The output buffer points to a data buffer containing the data to be sent. The IN direction is from the host perspective representing an outbound transfer from the device to the host.

## **Output buffer length**

The size of the data to be sent.

## **Status block**

If the request is successful, the USB function class extension (UFX) returns **STATUS\_SUCCESS**, or another status value for which **NT\_SUCCESS(status)** equals TRUE. Otherwise it returns a status value for which **NT\_SUCCESS(status)** equals FALSE.

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

UFX forwards this IOCTL request to the transfer queue created for the endpoint by [UfxEndpointCreate](#).

The function controller initiates a transfer in the IN direction on the endpoint and automatically appends a zero-length packet transfer after the data provided in the data buffer is successfully sent. A zero-length packet is only appended by the controller if the size of the transfer payload is a multiple of the endpoint's maximum packet size.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbfnioclt.h

# **IOCTL\_INTERNAL\_USBFN\_TRANSFER\_OUT IOCTL (usbfnioclt.h)**

Article02/22/2024

The class driver sends this request to initiate a data transfer from the host on the specified pipe.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a **USBFNPIPEID** type that specifies the pipe ID.

## **Input buffer length**

The size of a **USBFNPIPEID** type.

## **Output buffer**

A data buffer to receive data from the host.

## **Output buffer length**

The length of the buffer.

## **Status block**

`Irп->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful.

Otherwise, Status to the appropriate error condition as a `NTSTATUS` code.

For more information, see [NTSTATUS Values](#).

## **Remarks**

This request must be sent after sending the [IOCTL\\_INTERNAL\\_USBFN\\_ACTIVATE\\_USB\\_BUS](#) request.

# Requirements

 Expand table

Requirement	Value
Header	usbfniocctl.h

# USBFN\_POWER\_FILTER\_STATE structure (usbfnioctl.h)

Article 02/22/2024

Reserved. Do not use.

## Syntax

C++

```
typedef struct _USBFN_POWER_FILTER_STATE {
    USBFN_DEVICE_STATE DeviceState;
    union {
        ULONG PState;
        ULONG Reserved;
    };
} USBFN_POWER_FILTER_STATE, *PUSBFN_POWER_FILTER_STATE;
```

## Members

DeviceState

Describes the USB device states for the device/controller. These states correspond to the USB device states as defined in section 9.1 of the USB 2.0 Specification.

PState

USB device states for the device/controller.

Reserved

Do not use.

## Requirements

 Expand table

Requirement	Value
Header	usbfnioctl.h (include Usbfnioctl.h)

## See also

[IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_EXIT\\_LPM](#)

[IOCTL\\_INTERNAL\\_USBFN\\_SET\\_POWER\\_FILTER\\_STATE](#)

# usbioclt.h header

Article01/23/2023

This header is used by usbref. For more information, see:

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

usbioclt.h contains the following programming interfaces:

## IOCTLs

### [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#)

The IOCTL\_GET\_HCD\_DRIVERKEY\_NAME I/O control request retrieves the driver key name in the registry for a USB host controller driver.

### [IOCTL\\_INTERNAL\\_USB\\_CYCLE\\_PORT](#)

The IOCTL\_INTERNAL\_USB\_CYCLE\_PORT I/O request simulates a device unplug and replug on the port associated with the PDO.

### [IOCTL\\_INTERNAL\\_USB\\_ENABLE\\_PORT](#)

The IOCTL\_INTERNAL\_USB\_ENABLE\_PORT IOCTL has been deprecated. Do not use.

### [IOCTL\\_INTERNAL\\_USB\\_GET\\_BUS\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO I/O request queries the bus driver for certain bus information.

### [IOCTL\\_INTERNAL\\_USB\\_GET\\_BUSGUID\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_BUSGUID\_INFO IOCTL has been deprecated. Do not use.

### [IOCTL\\_INTERNAL\\_USB\\_GET\\_CONTROLLER\\_NAME](#)

The IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME I/O request queries the bus driver for the device name of the USB host controller.

### [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_CONFIG\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO I/O request returns information about a USB device and the hub it is attached to.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_HANDLE](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_HANDLE\\_EX](#)

The IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE\_EX IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_HUB\\_COUNT](#)

The IOCTL\_INTERNAL\_USB\_GET\_HUB\_COUNT IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_HUB\\_NAME](#)

The IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME I/O request is used by drivers to retrieve the UNICODE symbolic name for the target PDO if the PDO is for a hub.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_PARENT\\_HUB\\_INFO](#)

The IOCTL\_INTERNAL\_USB\_GET\_PARENT\_HUB\_INFO is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_PORT\\_STATUS](#)

The IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS I/O request queries the status of the PDO. IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS is a kernel-mode I/O control request. This request targets the USB hub PDO. This IOCTL must be sent at IRQL = PASSIVE\_LEVEL.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_ROOTHUB\\_PDO](#)

The IOCTL\_INTERNAL\_USB\_GET\_ROOTHUB\_PDO IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_TOPOLOGY\\_ADDRESS](#)

The IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS I/O request returns information about the host controller the USB device is attached to, and the device's location in the USB device tree.

## [IOCTL\\_INTERNAL\\_USB\\_GET\\_TT\\_DEVICE\\_HANDLE](#)

The IOCTL\_INTERNAL\_USB\_GET\_TT\_DEVICE\_HANDLE is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_NOTIFY\\_IDLE\\_READY](#)

The IOCTL\_INTERNAL\_USB\_NOTIFY\_IDLE\_READY IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_RECORD\\_FAILURE](#)

The IOCTL\_INTERNAL\_USB\_RECORD\_FAILURE IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

The IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE I/O request registers the driver of a USB multi-function device (composite driver) with the underlying USB driver stack.

## [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_RESUME](#)

The IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_RESUME IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_REQ\\_GLOBAL\\_SUSPEND](#)

The IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_SUSPEND IOCTL is used by the USB hub driver. Do not use.

## [IOCTL\\_INTERNAL\\_USB\\_REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#)

The IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION I/O request is sent by the driver of a Universal Serial Bus (USB) multi-function device (composite driver) to request remote wake-up notifications from a specific function in the device.

## [IOCTL\\_INTERNAL\\_USB\\_RESET\\_PORT](#)

The IOCTL\_INTERNAL\_USB\_RESET\_PORT I/O control request is used by a driver to reset the upstream port of the device it manages.

## [IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_IDLE\\_NOTIFICATION](#)

The IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION I/O request is used by drivers to inform the USB bus driver that a device is idle and can be suspended.

## [IOCTL\\_INTERNAL\\_USB\\_SUBMIT\\_URB](#)

The IOCTL\_INTERNAL\_USB\_SUBMIT\_URB I/O control request is used by drivers to submit an URB to the bus driver. IOCTL\_INTERNAL\_USB\_SUBMIT\_URB is a kernel-mode I/O control request. This request targets the USB hub PDO.

## [IOCTL\\_INTERNAL\\_USB\\_UNREGISTER\\_COMPOSITE\\_DEVICE](#)

The IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE I/O request unregisters the driver of a USB multi-function device (composite driver) and releases all resources that are associated with registration.

## [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_OFF](#)

The IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_OFF I/O control has been deprecated. Do not use.

## [IOCTL\\_USB\\_DIAG\\_IGNORE\\_HUBS\\_ON](#)

The IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_ON I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_OFF](#)

The IOCTL\_USB\_DIAGNOSTIC\_MODE\_OFF I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_DIAGNOSTIC\\_MODE\\_ON](#)

The IOCTL\_USB\_DIAGNOSTIC\_MODE\_ON I/O control has been deprecated. Do not use.

#### [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#)

The IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION I/O control request retrieves one or more descriptors for the device that is associated with the indicated port index. IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION is a user-mode I/O control request.

#### [IOCTL\\_USB\\_GET\\_DEVICE\\_CHARACTERISTICS](#)

The client driver sends this request to determine general characteristics about a USB device, such as maximum send and receive delays for any request.

#### [IOCTL\\_USB\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#)

The IOCTL\_USB\_GET\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC IOCTL function gets the system query performance counter (QPC) value for a specific frame and microframe.

#### [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES](#)

The IOCTL\_USB\_GET\_HUB\_CAPABILITIES I/O control request retrieves the capabilities of a USB hub.

#### [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES\\_EX](#)

The IOCTL\_USB\_GET\_HUB\_CAPABILITIES\_EX I/O control request retrieves the capabilities of a USB hub. IOCTL\_USB\_GET\_HUB\_CAPABILITIES\_EX is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

#### [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#)

The IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX I/O control request is sent by an application to retrieve information about a USB hub in a USB\_HUB\_INFORMATION\_EX structure. The request retrieves the highest port number on the hub.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES I/O control request retrieves the Microsoft-extended port attributes for a specific port.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME I/O control request retrieves the driver registry key name that is associated with the device that is connected to the indicated port.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION request retrieves information about the indicated USB port and the device that is attached to the port, if there is one.Client drivers must send this IOCTL at an IRQL of

PASSIVE\_LEVEL.IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB). Do not send this request to the root hub.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request retrieves information about a USB port and the device that is attached to the port, if there is one.Client drivers must send this IOCTL at an IRQL of PASSIVE\_LEVEL.IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB). Do not send this request to the root hub.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 I/O control is sent by an application to retrieve information about the protocols that are supported by a particular USB port on a hub. The request also retrieves the speed capability of the port.

#### [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_NAME](#)

The IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME I/O control request is used with the USB\_NODE\_CONNECTION\_NAME structure to retrieve the symbolic link name of the hub that is attached to the downstream port.IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

#### [IOCTL\\_USB\\_GET\\_NODE\\_INFORMATION](#)

The IOCTL\_USB\_GET\_NODE\_INFORMATION I/O control request is used with the USB\_NODE\_INFORMATION structure to retrieve information about a parent device.IOCTL\_USB\_GET\_NODE\_INFORMATION is a user-mode I/O control request.

#### [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

The IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request is sent by an application to retrieve information about a specific port on a USB hub.

#### [IOCTL\\_USB\\_GET\\_ROOT\\_HUB\\_NAME](#)

The IOCTL\_USB\_GET\_ROOT\_HUB\_NAME I/O control request is used with the USB\_ROOT\_HUB\_NAME structure to retrieve the symbolic link name of the root hub.IOCTL\_USB\_GET\_ROOT\_HUB\_NAME is a user-mode I/O control request.

## [IOCTL\\_USB\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

The client driver sends this request to retrieve the transport characteristics.

## [IOCTL\\_USB\\_HCD\\_DISABLE\\_PORT](#)

The IOCTL\_USB\_HCD\_DISABLE\_PORT IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_ENABLE\\_PORT](#)

The IOCTL\_USB\_HCD\_ENABLE\_PORT IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_GET\\_STATS\\_1](#)

The IOCTL\_USB\_HCD\_GET\_STATS\_1 IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HCD\\_GET\\_STATS\\_2](#)

The IOCTL\_USB\_HCD\_GET\_STATS\_2 IOCTL has been deprecated. Do not use.

## [IOCTL\\_USB\\_HUB\\_CYCLE\\_PORT](#)

The IOCTL\_USB\_HUB\_CYCLE\_PORT I/O control request power-cycles the port that is associated with the PDO that receives the request.

## [IOCTL\\_USB\\_NOTIFY\\_ON\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request notifies the caller of change in transport characteristics.

## [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request registers for notifications about the changes in transport characteristics.

## [IOCTL\\_USB\\_RESET\\_HUB](#)

The IOCTL\_USB\_RESET\_HUB IOCTL is used by the USB driver stack. Do not use.

## [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

This request registers the caller with USB driver stack for time sync services.

## [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

This request unregisters the caller with USB driver stack for time sync services.

## [IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

This request unregisters the caller from getting notifications about transport characteristics changes.

# Structures

## [HUB\\_DEVICE\\_CONFIG\\_INFO](#)

The HUB\_DEVICE\_CONFIG\_INFO structure is used in conjunction with the kernel-mode IOCTL, IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO to request to report information about a USB device and the hub to which the device is attached.

## [USB\\_BUS\\_NOTIFICATION](#)

Learn more about: [\\_USB\\_BUS\\_NOTIFICATION](#) structure

## [USB\\_CYCLE\\_PORT\\_PARAMS](#)

The USB\_CYCLE\_PORT\_PARAMS structure is used with the IOCTL\_USB\_HUB\_CYCLE\_PORT I/O control request to power cycle the port that is associated with the PDO that receives the request.

## [USB\\_DESCRIPTOR\\_REQUEST](#)

The USB\_DESCRIPTOR\_REQUEST structure is used with the IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION I/O control request to retrieve one or more descriptors for the device that is associated with the indicated connection index.

## [USB\\_DEVICE\\_CHARACTERISTICS](#)

Contains information about the USB device's characteristics, such as the maximum send and receive delays for any request. This structure is used in the IOCTL\_USB\_GET\_DEVICE\_CHARACTERISTICS request.

## [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

Stores the frame and microframe numbers and the calculated system QPC values. This structure is used in the IOCTL\_USB\_GET\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC request.

## [USB\\_HCD\\_DRIVERKEY\\_NAME](#)

The USB\_HCD\_DRIVERKEY\_NAME structure is used with the IOCTL\_GET\_HCD\_DRIVERKEY\_NAME I/O control request to retrieve the driver key in the registry for the USB host controller driver.

## [USB\\_HUB\\_CAP\\_FLAGS](#)

The USB\_HUB\_CAP\_FLAGS structure is used to report the capabilities of a hub.

## [USB\\_HUB\\_CAPABILITIES](#)

The USB\_HUB\_CAPABILITIES structure has been deprecated. Use [USB\\_HUB\\_CAPABILITIES\\_EX](#) instead.

## [USB\\_HUB\\_CAPABILITIES\\_EX](#)

The USB\_HUB\_CAPABILITIES\_EX structure is used with the IOCTL\_USB\_GET\_HUB\_CAPABILITIES I/O control request to retrieve the capabilities of a particular USB hub.

## [USB\\_HUB\\_INFORMATION](#)

The USB\_HUB\_INFORMATION structure contains information about a hub.

## [USB\\_HUB\\_INFORMATION\\_EX](#)

The USB\_HUB\_INFORMATION\_EX structure is used with the IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX I/O control request to retrieve information about a Universal Serial Bus (USB) hub.

## [USB\\_HUB\\_NAME](#)

The USB\_HUB\_NAME structure stores the hub's symbolic device name.

## [USB\\_ID\\_STRING](#)

The USB\_ID\_STRING structure is used to store a string or multi-string.

## [USB\\_MI\\_PARENT\\_INFORMATION](#)

The USB\_MI\_PARENT\_INFORMATION structure contains information about a composite device.

## [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

The USB\_NODE\_CONNECTION\_ATTRIBUTES structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES I/O control request to retrieve the attributes of a connection.

## [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)

The USB\_NODE\_CONNECTION\_DRIVERKEY\_NAME structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME I/O control request to retrieve the driver key name for the device that is connected to the indicated port.

## [USB\\_NODE\\_CONNECTION\\_INFORMATION](#)

The USB\_NODE\_CONNECTION\_INFORMATION structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION request to retrieve information about a USB port and connected device.

## [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX structure is used in conjunction with the

IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request to obtain information about the connection associated with the indicated USB port.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 I/O control request to retrieve speed information about a Universal Serial Bus (USB) device that is attached to a particular port.

#### [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2\\_FLAGS](#)

The USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2\_FLAGS union is used to indicate the speed at which a USB 3.0 device is currently operating and whether it can operate at higher speed, when attached to a particular port.

#### [USB\\_NODE\\_CONNECTION\\_NAME](#)

The USB\_NODE\_CONNECTION\_NAME structure is used with the IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME I/O control request to retrieve the symbolic link of the downstream hub that is attached to the port.

#### [USB\\_NODE\\_INFORMATION](#)

The USB\_NODE\_INFORMATION structure is used with the IOCTL\_USB\_GET\_NODE\_INFORMATION I/O control request to retrieve information about a parent device.

#### [USB\\_PIPE\\_INFO](#)

The USB\_PIPE\_INFO structure is used in conjunction with the USB\_NODE\_CONNECTION\_INFORMATION\_EX structure and the IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX request to obtain information about a connection and its associated pipes.

#### [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

The USB\_PORT\_CONNECTOR\_PROPERTIES structure is used with the IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request to retrieve information about a port on a particular SuperSpeed hub.

#### [USB\\_PORT\\_PROPERTIES](#)

The USB\_PORT\_PROPERTIES union is used to report the capabilities of a Universal Serial Bus (USB) port. The port capabilities are retrieved in the USB\_PORT\_CONNECTOR\_PROPERTIES structure by the IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES I/O control request.

#### [USB\\_PROTOCOLS](#)

The USB\_PROTOCOLS union is used to report the Universal Serial Bus (USB) signaling protocols that are supported by the port.

## [USB\\_ROOT\\_HUB\\_NAME](#)

The USB\_ROOT\_HUB\_NAME structure stores the root hub's symbolic device name.

## [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

The input and output buffer for the IOCTL\_USB\_START\_TRACKING\_FOR\_TIME\_SYNC request.

## [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#)

The input buffer for the IOCTL\_USB\_STOP\_TRACKING\_FOR\_TIME\_SYNC request.

## [USB\\_TOPOLOGY\\_ADDRESS](#)

The USB\_TOPOLOGY\_ADDRESS structure is used with the IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS I/O request to retrieve information about a USB device's location in the USB device tree.

## [USB\\_TRANSPORT\\_CHARACTERISTICS](#)

Stores the transport characteristics at relevant points in time. This structure is used in the IOCTL\_USB\_GET\_TRANSPORT\_CHARACTERISTICS request.

## [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#)

Contains registration information filled when the IOCTL\_USB\_REGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request completes.

## [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_REGISTRATION](#)

Contains registration information for the IOCTL\_USB\_REGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request.

## [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_UNREGISTRATION](#)

Contains unregistration information for the IOCTL\_USB\_UNREGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE request.

# Enumerations

## [USB\\_CONNECTION\\_STATUS](#)

The USB\_CONNECTION\_STATUS enumerator indicates the status of the connection to a device on a USB hub port.

## [USB\\_HUB\\_NODE](#)

The USB\_HUB\_NODE enumerator indicates whether a device is a hub or a composite device.

## [USB\\_HUB\\_TYPE](#)

The USB\_HUB\_TYPE enumeration defines constants that indicate the type of USB hub. The hub type is retrieved by the IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX I/O control request.

## [USB\\_NOTIFICATION\\_TYPE](#)

Learn more about: [\\_USB\\_NOTIFICATION\\_TYPE](#) enumeration

# HUB\_DEVICE\_CONFIG\_INFO structure (usbioc.h)

Article09/01/2022

The **HUB\_DEVICE\_CONFIG\_INFO** structure is used in conjunction with the kernel-mode IOCTL, [IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_CONFIG\\_INFO](#) to request to report information about a USB device and the hub to which the device is attached.

## Syntax

C++

```
typedef struct _HUB_DEVICE_CONFIG_INFO_V1 {
    ULONG             Version;
    ULONG             Length;
    USB_HUB_CAP_FLAGS HubFlags;
    USB_ID_STRING    HardwareIds;
    USB_ID_STRING    CompatibleIds;
    USB_ID_STRING    DeviceDescription;
    ULONG             Reserved[19];
    USB_HUB_DEVICE_UXD_SETTINGS UxdSettings;
} HUB_DEVICE_CONFIG_INFO, *PHUB_DEVICE_CONFIG_INFO;
```

## Members

**Version**

Specifies the version number. Must be set to 1.

**Length**

Specifies the size of the **HUB\_DEVICE\_CONFIG\_INFO** structure. Must be set by the caller.

**HubFlags**

Specifies the hub capabilities in a [USB\\_HUB\\_CAP\\_FLAGS](#) structure.

**HardwareIds**

The PnP hardware ID multi-string for the USB device in a [USB\\_ID\\_STRING](#) structure.

**CompatibleIds**

PnP compatible ID multi-string for the USB device in a [USB\\_ID\\_STRING](#) structure.

**DeviceDescription**

Description of the device in a [USB\\_ID\\_STRING](#) structure. This may be set to **NULL**.

**Reserved[19]**

Reserved.

**UxdSettings**

## Remarks

The **Buffer** member of the [USB\\_ID\\_STRING](#) structure points to a string that contains **HardwareIds**, **CompatibleIds**, and **DeviceDescription** values. The caller is responsible for releasing this string buffer, which is allocated by the hub driver.

## Requirements

<b>Minimum supported client</b>	Available in Windows XP and later operating systems.
<b>Header</b>	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_INTERNAL\\_USB\\_GET\\_DEVICE\\_CONFIG\\_INFO](#)

[USB Structures](#)

[USB\\_HUB\\_CAP\\_FLAGS](#)

[USB\\_ID\\_STRING](#)

# IOCTL\_GET\_HCD\_DRIVERKEY\_NAME

## IOCTL (usbioclt.h)

Article05/18/2021

The **IOCTL\_GET\_HCD\_DRIVERKEY\_NAME** I/O control request retrieves the driver key name in the registry for a USB host controller driver.

**IOCTL\_GET\_HCD\_DRIVERKEY\_NAME** is a user-mode I/O control request. This request targets the USB host controller (GUID\_DEVINTERFACE\_USB\_HOST\_CONTROLLER).

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Input buffer

None.

### Input buffer length

None.

### Output buffer

The **AssociatedIrp.SystemBuffer** member specifies the address of a caller-allocated buffer that contains a [USB\\_HCD\\_DRIVERKEY\\_NAME](#) structure. On output, this structure holds the driver key name. For more information, see Remarks.

### Output buffer length

The size of this buffer is specified in the **Parameters.DeviceloControl.OutputBufferLength** member.

### Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

# Remarks

To get the driver key name in the registry, you must perform the following tasks:

1. Declare a variable of the type [USB\\_HCD\\_DRIVERKEY\\_NAME](#).
2. Send an [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#) request by specifying the address and size of the variable in the output parameters. On return, the **ActualLength** member of [USB\\_HCD\\_DRIVERKEY\\_NAME](#) contains the length required to allocate a buffer to hold a [USB\\_HCD\\_DRIVERKEY\\_NAME](#) that is populated with the driver key name.
3. Allocate memory for a buffer to hold a [USB\\_HCD\\_DRIVERKEY\\_NAME](#) structure. The size of the buffer must be the received **ActualLength** value.
4. Send an [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#) request by passing a pointer to the allocated buffer and its size in the output parameters. On return, the **DriverKeyName** member of [USB\\_HCD\\_DRIVERKEY\\_NAME](#) is a null-terminated Unicode string that contains the name of the driver key associated with the host controller driver.

The following example code shows how to send the [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#) I/O control request.

C++

```
/*++

Routine Description:

This routine prints the name of the driver key associated with
the specified host controller driver.

Arguments:

HCD - Handle for host controller driver.

Return Value: Boolean that indicates success or failure.

--*/



BOOL GetHCDDriverKeyName (HANDLE HCD)
{
    BOOL             success;
    ULONG            nBytes;
    USB_HCD_DRIVERKEY_NAME  driverKeyName;
    PUSB_HCD_DRIVERKEY_NAME driverKeyNameW;

    driverKeyNameW = NULL;
```

```

// 1. Get the length of the name of the driver key.
success = DeviceIoControl(HCD,
    IOCTL_GET_HCD_DRIVERKEY_NAME,
    NULL,
    0,
    &driverKeyName,
    sizeof(driverKeyName),
    &nBytes,
    NULL);

if (!success)
{
    printf("First IOCTL_GET_HCD_DRIVERKEY_NAME request failed\n");
    goto GetHCDDriverKeyNameDone;
}

//2. Get the length of the driver key name.
nBytes = driverKeyName.ActualLength;

if (nBytes <= sizeof(driverKeyName))
{
    printf("Incorrect length received by
IOCTL_GET_HCD_DRIVERKEY_NAME.\n");
    goto GetHCDDriverKeyNameDone;
}

// 3. Allocate memory for a USB_HCD_DRIVERKEY_NAME
//      to hold the driver key name.
driverKeyNameW = (PUSB_HCD_DRIVERKEY_NAME) malloc(nBytes);

if (driverKeyNameW == NULL)
{
    printf("Failed to allocate memory.\n");
    goto GetHCDDriverKeyNameDone;
}

// Get the name of the driver key of the device attached to
// the specified port.
success = DeviceIoControl(HCD,
    IOCTL_GET_HCD_DRIVERKEY_NAME,
    NULL,
    0,
    driverKeyNameW,
    nBytes,
    &nBytes,
    NULL);

if (!success)
{
    printf("Second IOCTL_GET_HCD_DRIVERKEY_NAME request failed.\n");
    goto GetHCDDriverKeyNameDone;
}

// print the driver key name.
printf("Driver Key Name: %s.\n", driverKeyNameW->DriverKeyName);

```

```
GetHCDDriverKeyNameDone:

    // Cleanup.
    // Free the allocated memory for USB_HCD_DRIVERKEY_NAME.

    if (driverKeyNameW != NULL)
    {
        free(driverKeyNameW);
        driverKeyNameW = NULL;
    }

    return success;
}
```

## Requirements

Header	usbiocnl.h (include Usbiocnl.h)
--------	---------------------------------

## See also

[USB\\_HCD\\_DRIVERKEY\\_NAME](#)

# **IOCTL\_INTERNAL\_USB\_CYCLE\_PORT**

## **IOCTL (usbioclt.h)**

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_CYCLE\_PORT** I/O request simulates a device unplug and replug on the port associated with the PDO.

Drivers should cancel all I/O requests and wait for them to complete before initiating this operation.

A driver that manages an individual interface on a composite device cannot cycle the port to which the device is attached without affecting the entire composite device and all of its interfaces. For this reason, drivers that manage interfaces should attempt other types of error recovery, such as resetting pipes ([\\_URB\\_PIPE\\_REQUEST](#)), before cycling the port.

**IOCTL\_INTERNAL\_USB\_CYCLE\_PORT** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of **PASSIVE\_LEVEL**.

### **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### **Input buffer**

None.

### **Input buffer length**

None.

### **Output buffer**

None.

### **Output buffer length**

None.

## Status block

The bus or port driver sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` or the appropriate error status.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows XP and later operating systems.
Header	<code>usbioctl.h</code> (include <code>Usbioctl.h</code> )

# IOCTL\_INTERNAL\_USB\_ENABLE\_PORT

## IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_ENABLE\_PORT** IOCTL has been deprecated. Do not use.

### Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Supported in Microsoft Windows 2000, Windows XP and Windows Server 2003. It is not supported in Windows Vista and later operating systems.
Header	usioctl.h (include Usioctl.h)

# IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO

## IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO** I/O request queries the bus driver for certain bus information.

**IOCTL\_INTERNAL\_USB\_GET\_BUS\_INFO** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of **PASSIVE\_LEVEL**.

### Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### Input buffer

**Parameters.Others.Argument1** should be a pointer to a [USB\\_BUS\\_NOTIFICATION](#) structure.

### Input buffer length

The size of a [USB\\_BUS\\_NOTIFICATION](#) structure.

### Output buffer

**Parameters.Others.Argument1** points to a [USB\\_BUS\\_NOTIFICATION](#) structure that has the **TotalBandwidth**, **ConsumedBandwidth**, and **ControllerNameLength** fields filled in.

### Output buffer length

The size of a [USB\\_BUS\\_NOTIFICATION](#) structure.

### Status block

The bus or port driver sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** or the appropriate error status.

# Requirements

 [Expand table](#)

Requirement	Value
Header	usbioclt.h (include Usbioclt.h)

## See also

[USB\\_BUS\\_NOTIFICATION](#)

# IOCTL\_INTERNAL\_USB\_GET\_BUSGUID\_INFO IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_BUSGUID\_INFO** IOCTL has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usbioclt.h (include Usbloctl.h)

# **IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME IOCTL (usbioctl.h)**

Article 02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME** I/O request queries the bus driver for the device name of the USB host controller.

**IOCTL\_INTERNAL\_USB\_GET\_CONTROLLER\_NAME** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of **PASSIVE\_LEVEL**.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

**Parameters.Others.Argument1** should be a pointer to a [USB\\_HUB\\_NAME](#) structure that will be filled in with the name of the host controller.

## **Input buffer length**

**Parameters.Others.Argument2** should be a ULONG specifying the length of the buffer (in bytes) in **Parameters.Others.Argument1**.

## **Output buffer**

The bus driver will fill the buffer pointed to by **Parameters.Others.Argument1** with the host controller device name.

## **Output buffer length**

It will be filled only up to the length specified in **Parameters.Others.Argument2**.

## **Status block**

The bus or port driver sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** or the appropriate error status.

## Remarks

The caller must supply a buffer that is large enough to hold a [USB\\_HUB\\_NAME](#) structure. The **Parameters.Others.Argument2** value indicates size of that buffer. Upon successful completion, the **HubName** member of **USB\_HUB\_NAME** contains the name of the controller and the **ActualLength** member indicates the length of the controller name string. Note that **ActualLength** does not indicate the size of the entire **USB\_HUB\_NAME** structure. If the buffer supplied in **Parameters.Others.Argument1** is not large enough to hold the string, the **HubName** value might show a truncated string.

To obtain the size of the buffer required to hold the string, send the request twice. In the first request, specify a buffer that is at least `sizeof(USB_HUB_NAME)` bytes. Otherwise, **ActualLength** does not indicate the correct length of the string and the request fails with STATUS\_BUFFER\_TOO\_SMALL.

After the first request completes successfully, allocate a buffer of `ActualLength + sizeof(ULONG)` bytes and send the request again. After the request completes, **HubName** indicates the entire controller name string.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioclt.h

## See also

[USB\\_HUB\\_NAME](#)

# IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO IOCTL (usbioc.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO** I/O request returns information about a USB device and the hub it is attached to.

**IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_CONFIG\_INFO** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of DISPATCH\_LEVEL or lower.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

**Parameters.Others.Argument1** points to a [HUB\\_DEVICE\\_CONFIG\\_INFO](#) structure to receive the device configuration information.

## Input buffer length

The size of a [HUB\\_DEVICE\\_CONFIG\\_INFO](#) structure.

## Output buffer

**Parameters.Others.Argument1** points to a [HUB\\_DEVICE\\_CONFIG\\_INFO](#) structure containing the device configuration information.

## Output buffer length

The size of a [HUB\\_DEVICE\\_CONFIG\\_INFO](#) structure.

## Status block

The hub or port driver sets **Irp->IoStatus.Status** to STATUS\_SUCCESS or the appropriate error status.

## Remarks

Upon successful completion, the HardwareIds, CompatibleIds, DeviceDescription [USB\\_ID\\_STRING](#) structures contained in the [HUB\\_DEVICE\\_CONFIG\\_INFO](#) structure points to string buffers allocated by the hub driver. The caller driver is responsible for releasing these buffers before the driver unloads.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista and later operating systems.
Header	usbiocrtl.h (include Usbiocrtl.h)

## See also

[HUB\\_DEVICE\\_CONFIG\\_INFO](#)

[USB\\_ID\\_STRING](#)

# IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE IOCTL (usioctl.h)

Article02/22/2024

The `IOCTL_INTERNAL_USB_GET_DEVICE_HANDLE` IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista and later operating systems.
Header	usioctl.h (include Usioctl.h)

# IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE\_EX IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_DEVICE\_HANDLE\_EX** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista and later operating systems.
Header	usioctl.h (include Usioctl.h)

# IOCTL\_INTERNAL\_USB\_GET\_HUB\_COUN T IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_HUB\_COUNT** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbioclt.h (include Usbioclt.h)

# **IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME**

## **IOCTL (usbioclt.h)**

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME** I/O request is used by drivers to retrieve the UNICODE symbolic name for the target PDO if the PDO is for a hub. Otherwise a NULL string is returned.

Drivers can use the symbolic name to retrieve additional information about the hub through user-mode I/O control requests and WMI calls.

**IOCTL\_INTERNAL\_USB\_GET\_HUB\_NAME** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of **PASSIVE\_LEVEL**.

### **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### **Input buffer**

`Irp->AssociatedIrp.SystemBuffer` points to a [USB\\_HUB\\_NAME](#) structure.

### **Input buffer length**

`Parameters.DeviceloControl.OutputBufferLength` is the length of the buffer (in bytes) passed in the `Irp->AssociatedIrp.SystemBuffer` field.

### **Output buffer**

`Irp->AssociatedIrp.SystemBuffer` is filled with the root hub's symbolic name.

### **Output buffer length**

The length of the root hub's symbolic name.

### **Status block**

A lower-level driver sets **Irp->IoStatus.Status** to STATUS\_SUCCESS or the appropriate error status. It will set **Irp->IoStatus.Information** to the number of bytes required to hold the **USB\_ROOT\_HUB\_NAME** structure. If the request fails, the driver can use this information to resubmit the request with a big enough buffer.

## Requirements

 Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB\\_ROOT\\_HUB\\_NAME](#)

# IOCTL\_INTERNAL\_USB\_GET\_PARENT\_HUB\_INFO IOCTL (usbioc.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_PARENT\_HUB\_INFO** is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usbioc.h (include Usbioc.h)

# IOCTL\_INTERNAL\_USB\_GET\_PORT\_STAT US IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS** I/O request queries the status of the PDO.

**IOCTL\_INTERNAL\_USB\_GET\_PORT\_STATUS** is a kernel-mode I/O control request. This request targets the USB hub PDO. This IOCTL must be sent at IRQL = PASSIVE\_LEVEL.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

**Parameters.Others.Argument1** should be a pointer to a ULONG to be filled in with the port status flags.

## Input buffer length

The size of a ULONG.

## Output buffer

**Parameters.Others.Argument1** points to a ULONG that has the port status flags filled in. The flags can be one or both of USBD\_PORT\_ENABLED (bit 0) or USBD\_PORT\_CONNECTED (bit 1). When the USB\_PORT\_ENABLED bit is set, the port has been enabled after resetting the device connected to the port. When the USB\_PORT\_ENABLED bit is clear, software has disabled the port or hardware has disabled it due to abnormal hardware conditions. When the USB\_PORT\_CONNECTED bit is set, the host controller root hub or external hub has detected that a device is connected to the port. When the USB\_PORT\_CONNECTED bit is clear, the host controller root hub or external hub has detected that a device is not connected to the port.

## Output buffer length

The size of a ULONG.

## Status block

The bus or port driver sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` or the appropriate error status.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Header	usbioclt.h (include Usbioclt.h)

# IOCTL\_INTERNAL\_USB\_GET\_ROOTHUB\_PDO IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_ROOTHUB\_PDO** IOCTL is used by the USB hub driver.  
Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usioctl.h

# IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS IOCTL (usbioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS** I/O request returns information about the host controller the USB device is attached to, and the device's location in the USB device tree.

**IOCTL\_INTERNAL\_USB\_GET\_TOPOLOGY\_ADDRESS** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of DISPATCH\_LEVEL or lower.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

Parameters.Others.Argument1 points to a [USB\\_TOPOLOGY\\_ADDRESS](#) structure to receive the device topology information.

## Input buffer length

The size of a [USB\\_TOPOLOGY\\_ADDRESS](#) structure.

## Output buffer

Parameters.Others.Argument1 points to a [USB\\_TOPOLOGY\\_ADDRESS](#) structure containing the device topology information.

## Output buffer length

The size of a [USB\\_TOPOLOGY\\_ADDRESS](#) structure.

## Status block

The hub or port driver sets `Irp->IoStatus.Status` to STATUS\_SUCCESS or the appropriate error status.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows Vista and later operating systems.
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB\\_TOPOLOGY\\_ADDRESS](#)

# IOCTL\_INTERNAL\_USB\_GET\_TT\_DEVICE\_HANDLE IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_GET\_TT\_DEVICE\_HANDLE** is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usioctl.h

# IOCTL\_INTERNAL\_USB\_NOTIFY\_IDLE\_READY IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_NOTIFY\_IDLE\_READY** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usioctl.h

# IOCTL\_INTERNAL\_USB\_RECORD\_FAILURE IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_RECORD\_FAILURE** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[+] Expand table

Requirement	Value
Header	usioctl.h

# IOCTL\_INTERNAL\_USB\_REGISTER\_COMP OSITE\_DEVICE IOCTL (usbioclt.h)

Article06/16/2023

The **IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE** I/O request registers the driver of a USB multi-function device (composite driver) with the underlying USB driver stack.

This request is sent by a driver that replaces the Microsoft-provided composite driver, Usbccgp.sys, and implements the function suspend and remote wake-up feature, per the Universal Serial Bus (USB) 3.0 specification.

**IOCTL\_INTERNAL\_USB\_REGISTER\_COMPOSITE\_DEVICE** is a kernel-mode I/O control request. This request targets the USB hub physical device object (PDO). This request must be sent at an interrupt request level (IRQL) of PASSIVE\_LEVEL.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

**Parameters.Others.Argument1** is a pointer to a caller-allocated and initialized [REGISTER\\_COMPOSITE\\_DEVICE](#) structure that contains information about the parent driver. To initialize the structure, call the [USBD\\_BuildRegisterCompositeDevice](#) routine.

The **AssociatedIrp.SystemBuffer** member points to a caller-allocated buffer that is large enough to hold an array of function handles (typed USBD\_FUNCTION\_HANDLE) for functions in the USB composite device. The number of elements in the array is indicated by the **FunctionCount** member of [REGISTER\\_COMPOSITE\\_DEVICE](#). To obtain the number of functions, inspect the descriptors returned by a get-configuration request.

## Input buffer length

The size of a [REGISTER\\_COMPOSITE\\_DEVICE](#) structure.

## Output buffer

On output, the buffer pointed to by **AssociatedIrp.SystemBuffer** member is filled with function handles for functions in the multi-function device.

## Output buffer length

The size of function handles for functions in the device.

## Status block

The USB driver stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request completes successfully. `STATUS_SUCCESS` indicates that the function handles are valid.

In case of an error, `Irp->IoStatus.Status` contains an appropriate error status. For example, if the composite driver sends the request more than once, the `Status` is set to `STATUS_INVALID_DEVICE_REQUEST`.

## Remarks

The purpose of `IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DEVICE` is for the composite driver to send a registration request to the USB driver stack. In the registration request, the composite driver specifies the number of functions supported by the device. Therefore, you must send the `IOCTL_INTERNAL_USB_REGISTER_COMPOSITE_DEVICE` request after determining the number of functions. Typically, that information is retrieved in the composite driver's start-device routine `IRP_MN_START_DEVICE`. Note that number of interfaces in a configuration *can* indicate the number of functions, but not always. Certain multi-function devices expose multiple interfaces related to one function. To obtain the number of functions, you must inspect various descriptors that are related to a particular configuration. Those descriptors can be obtained through a get-descriptor request.

In response to the registration request, the USB driver stack provides a list of handles for the functions in the device. For a code example, see [How to Register a Composite Device](#).

After the composite driver is registered, the driver can configure the remote wake-up feature. By using the function handle, the composite driver can send a request `IOCTL_INTERNAL_USB_REQUEST_REMOTE_WAKE_NOTIFICATION` to get remote wake-up notifications from the USB driver stack, when the associated function sends a resume signal.

In order to remove the composite driver's association with the USB driver stack and release all resources that are allocated for registration, the driver must send the `IOCTL_INTERNAL_USB_UNREGISTER_COMPOSITE_DEVICE` request.

# Requirements

Minimum supported client	Windows 8
Header	usbioclt.h (include Usbioclt.h)
IRQL	PASSIVE_LEVEL

## See also

[How to Register a Composite Device](#)

[IOCTL\\_INTERNAL\\_USB\\_UNREGISTER\\_COMPOSITE\\_DEVICE](#)

# IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_RESUME IOCTL (usbioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_RESUME** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usbioctl.h

# IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_SUSPEND IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_REQ\_GLOBAL\_SUSPEND** IOCTL is used by the USB hub driver. Do not use.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Header	usioctl.h

# **IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION IOCTL (usbioclt.h)**

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION** I/O request is sent by the driver of a Universal Serial Bus (USB) multi-function device (composite driver) to request remote wake-up notifications from a specific function in the device.

**IOCTL\_INTERNAL\_USB\_REQUEST\_REMOTE\_WAKE\_NOTIFICATION** is a kernel-mode I/O control request. This request targets the USB hub physical device object (PDO).

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

Parameters.Others.Argument1 points to a caller-allocated and initialized [REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#) structure that contains information about the function whose resume signal the driver is interested in. That information includes the function handle and the interface with which the function is associated.

## **Input buffer length**

The size of a [REQUEST\\_REMOTE\\_WAKE\\_NOTIFICATION](#) structure.

## **Output buffer**

None.

## **Output buffer length**

None.

## **Status block**

The hub or port driver sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` or the appropriate error status.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8
Header	usbiocrtl.h (include Usbiocrtl.h)
IRQL	DISPATCH_LEVEL

## See also

[How to Implement Function Suspend in a Composite Driver](#)

# **IOCTL\_INTERNAL\_USB\_RESET\_PORT**

## **IOCTL (usbioclt.h)**

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_RESET\_PORT** I/O control request is used by a driver to reset the upstream port of the device it manages. After a successful reset, the bus driver reselects the configuration and any alternative interface settings that the device had before the reset occurred. All pipe handles, configuration handles and interface handles remain valid.

Drivers should cancel all I/O requests and wait for them to complete before initiating this operation.

A driver that manages an individual interface on a composite device cannot reset the interface without resetting the entire composite device and all of its interfaces. For this reason, drivers that manage interfaces should attempt other types of error recovery, such as resetting pipes ([\\_URB\\_PIPE\\_REQUEST](#)), before resetting the interface.

This IOCTL must be sent at an IRQL of PASSIVE\_LEVEL.

**IOCTL\_INTERNAL\_USB\_RESET\_PORT** is a kernel-mode I/O control request. This request targets the USB hub PDO.

### **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### **Input buffer**

None.

### **Input buffer length**

None.

### **Output buffer**

None.

### **Output buffer length**

None.

## Status block

The bus or port driver sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` or the appropriate error status.

## Requirements

[ ] Expand table

Requirement	Value
Header	<code>usbiocrtl.h</code> (include <code>Usbiocrtl.h</code> )

## See also

[\\_URB\\_PIPE\\_REQUEST](#)

# IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION IOCTL (usbioctl.h)

Article 02/22/2024

The **IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION** I/O request is used by drivers to inform the USB bus driver that a device is idle and can be suspended.

When sending this IOCTL, caller must furnish a callback routine that does the actual suspension of the device. The USB bus driver will call this routine at PASSIVE\_LEVEL when it is safe for the device to be powered down. If the device supports remote wake up and has no Wait/Wake IRP already pending, the callback routine should submit a Wait/Wake IRP to the bus driver for the device, before powering it down.

For additional information, see [Supporting Devices that Have Wake-Up Capabilities](#) and [USB Selective Suspend](#).

**IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION** is a kernel-mode I/O control request. This request targets the USB hub PDO. This request must be sent at an IRQL of PASSIVE\_LEVEL.

## Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## Input buffer

Parameters.**DeviceIoControl.Type3InputBuffer** should be a pointer to a **USB\_IDLE\_CALLBACK\_INFO** structure. This structure should contain a pointer to the callback routine and a pointer to the callback routine context.

The structure holding the callback information is defined in usbioctl.h as follows:

C++

```
typedef VOID (*USB_IDLE_CALLBACK)(PVOID Context);

typedef struct _USB_IDLE_CALLBACK_INFO
{
    USB_IDLE_CALLBACK IdleCallback;
    PVOID IdleContext;
} USB_IDLE_CALLBACK_INFO, *PUSB_IDLE_CALLBACK_INFO;
```

## Input buffer length

The size of a `USB_IDLE_CALLBACK_INFO` structure.

## Output buffer

None.

## Output buffer length

None.

## Status block

The bus or port driver sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` or the appropriate error status.

## Requirements

[Expand table](#)

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

# IOCTL\_INTERNAL\_USB\_SUBMIT\_URB

## IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_SUBMIT\_URB** I/O control request is used by drivers to submit an [URB](#) to the bus driver.

**IOCTL\_INTERNAL\_USB\_SUBMIT\_URB** is a kernel-mode I/O control request. This request targets the USB hub PDO.

### Major code

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

### Input buffer

**Parameters.Others.Argument1** points to the URB, a variable-length structure. The **UrbHeader.Function** member of the URB specifies the URB type. The length of URB, as well as the meaning of any additional members depends on the value of **UrbHeader.Function**. See [URB](#) for details.

### Input buffer length

The **UrbHeader.Length** member specifies the size in bytes of the URB.

### Output buffer

**Parameters.Others.Argument1** points to the [URB](#) structure. The **UrbHeader.Status** contains a USB status code for the requested operation. Any additional output depends on the **UrbHeader.Function** member of the URB submitted. See [URB](#) for details.

### Output buffer length

The **UrbHeader.Length** member specifies the size in bytes of the URB.

### Status block

The lower-level drivers will set **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the URB can be successfully processed. Otherwise, the bus driver will set it to the appropriate error

condition, such as STATUS\_INVALID\_PARAMETER, or STATUS\_INSUFFICIENT\_RESOURCES.

## Requirements

 Expand table

Requirement	Value
Header	usbioclt.h (include Usbioclt.h)

## See also

[URB](#)

# **IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE IOCTL (usbioc.h)**

Article02/22/2024

The **IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE** I/O request unregisters the driver of a USB multi-function device (composite driver) and releases all resources that are associated with registration. The request is successful only if the composite driver was previously registered with the underlying USB driver stack through the [IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#) request.

**IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE** is a kernel-mode I/O control request. This request targets the Universal Serial Bus (USB) hub physical device object (PDO). This request must be sent at an interrupt request level (IRQL) of PASSIVE\_LEVEL.

## **Major code**

[IRP\\_MJ\\_INTERNAL\\_DEVICE\\_CONTROL](#)

## **Input buffer**

None.

## **Input buffer length**

None.

## **Output buffer**

None.

## **Output buffer length**

None.

## **Status block**

The USB driver stack sets `Irp->IoStatus.Status` to STATUS\_SUCCESS or the appropriate error status.

## Remarks

You must send the IOCTL\_INTERNAL\_USB\_UNREGISTER\_COMPOSITE\_DEVICE request in the composite driver's remove-device ([IRP\\_MN\\_REMOVE\\_DEVICE](#)) routine.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 8
Header	usbiocrtl.h (include Usbiocrtl.h)
IRQL	PASSIVE_LEVEL

## See also

[How to Register a Composite Device](#)

[IOCTL\\_INTERNAL\\_USB\\_REGISTER\\_COMPOSITE\\_DEVICE](#)

# IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_OFF

## IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_OFF** I/O control has been deprecated. Do not use.

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Status block

Irп->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Supported on Microsoft Windows 2000 only.
Header	usbioclt.h (include Usbioclt.h)

# IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_ON IOCTL (usbioc.h)

Article02/22/2024

The **IOCTL\_USB\_DIAG\_IGNORE\_HUBS\_ON** I/O control has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Supported on Microsoft Windows 2000 only.
Header	usbioc.h (include Usbioc.h)

# IOCTL\_USB\_DIAGNOSTIC\_MODE\_OFF

## IOCTL (usbioclt.h)

Article02/22/2024

The `IOCTL_USB_DIAGNOSTIC_MODE_OFF` I/O control has been deprecated. Do not use.

### Major code

`IRP_MJ_DEVICE_CONTROL`

### Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful.

Otherwise, `Status` to the appropriate error condition as a `NTSTATUS` code.

For more information, see [NTSTATUS Values](#).

## Requirements

  [Expand table](#)

Requirement	Value
Header	<code>usbioclt.h</code> (include <code>Usbioclt.h</code> )

# IOCTL\_USB\_DIAGNOSTIC\_MODE\_ON

## IOCTL (usbioc.h)

Article02/22/2024

The IOCTL\_USB\_DIAGNOSTIC\_MODE\_ON I/O control has been deprecated. Do not use.

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

  [Expand table](#)

Requirement	Value
Header	usbioc.h (include Usbioc.h)

# IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION IOCTL (usbioctl.h)

Article02/22/2024

The **IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION** I/O control request retrieves one or more descriptors for the device that is associated with the indicated port index.

**IOCTL\_USB\_GET\_DESCRIPTOR\_FROM\_NODE\_CONNECTION** is a user-mode I/O control request. This request targets the USB hub device ([GUID\\_DEVINTERFACE\\_USB\\_HUB](#)).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

The **AssociatedIrp.SystemBuffer** member points to a [USB\\_DESCRIPTOR\\_REQUEST](#) structure that describes the descriptor request.

## Input buffer length

The **Parameters.DeviceIoControl.OutputBufferLength** member indicates the size, in bytes, of the user-allocated output buffer in the **Data** member of the [USB\\_DESCRIPTOR\\_REQUEST](#) structure.

## Output buffer

The **Data** member of the [USB\\_DESCRIPTOR\\_REQUEST](#) structure at **AssociatedIrp.SystemBuffer** points to the output buffer.

## Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

## Requirements

[ ] Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB\\_DESCRIPTOR\\_REQUEST GUID\\_DEVINTERFACE\\_USB\\_HUB](#)

# IOCTL\_USB\_GET\_DEVICE\_CHARACTERIST ICS IOCTL (usbioclt.h)

Article02/22/2024

The client driver sends this request to determine general characteristics about a USB device, such as maximum send and receive delays for any request.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input / Output buffer

The `AssociatedIrp.SystemBuffer` member is a pointer to a caller-allocated `USB_DEVICE_CHARACTERISTICS`s structure. On input, set `Version` to `USB_DEVICE_CHARACTERISTICS_VERSION_1`. On output `Version` is reset to a version less than or equal to `USB_DEVICE_CHARACTERISTICS_VERSION_1`; `UsbDeviceCharacteristicsFlags` is set to `USB_DEVICE_CHARACTERISTICS_MAXIMUM_PATH_DELAYS_AVAILABLE` and the remaining members of the structure is filled with delay information.

## Input / Output buffer length

The size of the `USB_DEVICE_CHARACTERISTICS` structure.

## Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates the appropriate error condition as a `NTSTATUS` code.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioclt.h
IRQL	<=DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

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

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_USB\_GET\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC IOCTL (usbioclt.h)**

Article07/28/2021

Retrieves the system query performance counter (QPC) value synchronized with the frame and microframe.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input / Output buffer**

A pointer to a [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure that contains the time tracking handled retrieved by the [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request. On input, the caller can specify a frame and microframe number for which to retrieve the QPC value.

On output, the **CurrentQueryPerformanceCounter** member is set to a value predicted by the USB driver stack. The value represents the system QPC value in microseconds.

## **Input / Output buffer length**

The size of the [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## **Status block**

**Irp->IoStatus.Status** is set to **STATUS\_SUCCESS** if the request is successful. Otherwise, **Status** indicates an the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

On input, the caller can optionally specify a frame and microframe for which the caller is interested in knowing the associated system QPC value. Those values must be provided in the **InputFrameNumber** and **InputMicroFrameNumber** members of [USB\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#), respectively. On output, the USB driver stack fills the

`QueryPerformanceCounterAtInputFrameOrMicroFrame` member with a system QPC value calculated by the USB driver stack.

If the caller is not interested those values, `InputFrameNumber` and `InputMicroFrameNumber` values must be initialized to 0. On output, `QueryPerformanceCounterAtInputFrameOrMicroFrame` is set to 0.

If the USB driver stack encountered a frame boundary, the `PredictedAccuracyInMicroSeconds` value indicates accuracy in 125-microseconds unit. It also takes into consideration if sufficient time has elapsed since tracking is enabled.

The USB driver stack can also predict the system QPC value that are synchronized with bus frame and microframe numbers retrieved directly from the host controller.

In order to predict QPC values with accuracy, the USB driver stack might poll the frame and microframe time sources. That polling operation might require raising or lowering of IRQL to eliminate scheduling delays interfering with the accuracy of the value (after attempting to read the register/QPC timer from passive IRQL for a given number of tries). Given this possible additional CPU cost, this interface must only be used to get associated USB bus and QPC values and must not be used as a replacement to existing methods for retrieving just the USB bus time.

## Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioclt.h
IRQL	<= DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# IOCTL\_USB\_GET\_HUB\_CAPABILITIES

## IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_USB\_GET\_HUB\_CAPABILITIES** I/O control request retrieves the capabilities of a USB hub. Note This request is replaced by [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES\\_EX](#) in Windows Vista.

**IOCTL\_USB\_GET\_HUB\_CAPABILITIES** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Input buffer

None.

### Input buffer length

None.

### Output buffer

The **AssociatedIrp.SystemBuffer** member points to a user-allocated [USB\\_HUB\\_CAPABILITIES](#) structure that describes the hub capabilities.

### Output buffer length

The **Parameters.DeviceIoControl.OutputBufferLength** member indicates the size, in bytes, of the output buffer in **SystemBuffer**. The output buffer size must be `>= sizeof(USB_HUB_CAPABILITIES)`.

### Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

# Requirements

 Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES\\_EX](#)

[USB\\_HUB\\_CAPABILITIES](#)

# IOCTL\_USB\_GET\_HUB\_CAPABILITIES\_EX

## IOCTL (usbioclt.h)

Article 02/22/2024

The `IOCTL_USB_GET_HUB_CAPABILITIES_EX` I/O control request retrieves the capabilities of a USB hub.

`IOCTL_USB_GET_HUB_CAPABILITIES_EX` is a user-mode I/O control request. This request targets the USB hub device (`GUID_DEVINTERFACE_USB_HUB`).

### Major code

`IRP_MJ_DEVICE_CONTROL`

### Input buffer

The `AssociatedIrp.SystemBuffer` member points to a user-allocated buffer.

### Input buffer length

The buffer length equals `sizeof(USB_HUB_CAPABILITIES_EX)`.

### Output buffer

`AssociatedIrp.SystemBuffer` points to a user-allocated `USB_HUB_CAPABILITIES_EX` structure. On output, this structure describes the hub capabilities.

### Output buffer length

The `Parameters.DeviceIoControl.OutputBufferLength` member indicates the size, in bytes, of the data that is returned at `SystemBuffer`, or `sizeof(USB_HUB_CAPABILITIES_EX)`.

### Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`. If the hub has not been started or is not functional, the request returns `STATUS_UNSUCCESSFUL`.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows Vista and later.
Header	usbioctl.h (include Usbiocctl.h)

## See also

[USB\\_HUB\\_CAPABILITIES\\_EX](#)

# IOCTL\_USB\_GET\_HUB\_INFORMATION\_E

## X IOCTL (usbioctl.h)

Article02/22/2024

The **IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX** I/O control request is sent by an application to retrieve information about a USB hub in a [USB\\_HUB\\_INFORMATION\\_EX](#) structure.

The request retrieves the highest port number on the hub. For USB 2.0 and SuperSpeed hubs (non-root hubs), the request also retrieves the associated hub descriptors, as defined in USB 2.0 and 3.0 Specifications, respectively.

**IOCTL\_USB\_GET\_HUB\_INFORMATION\_EX** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Input buffer

**AssociatedIrp.SystemBuffer** points to a caller-allocated [USB\\_HUB\\_INFORMATION\\_EX](#) structure.

### Input buffer length

The **Parameters.DeviceIoControl.InputBufferLength** member indicates the size, in bytes, of the caller-allocated buffer whose size equals `sizeof(USB_HUB_INFORMATION_EX)`.

### Output buffer

On output, the [USB\\_HUB\\_INFORMATION\\_EX](#) structure that is pointed to by **AssociatedIrp.SystemBuffer** is filled with information about the hub.

### Output buffer length

The **Parameters.DeviceIoControl.OutputBufferLength** member indicates the size, in bytes, of the output buffer **SystemBuffer**.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB driver stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	<code>usbioclt.h</code> (include <code>Usbioclt.h</code> )

## See also

[USB\\_HUB\\_INFORMATION\\_EX](#)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES** I/O control request retrieves the Microsoft-extended port attributes for a specific port.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_ATTRIBUTES** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

The **AssociatedIrp.SystemBuffer** member points to a user-allocated [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) structure. On input, the caller specifies the port number in the **ConnectionIndex** member of a [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) structure.

## Input buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) structure.

## Output buffer

On output, the [USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) structure at **AssociatedIrp.SystemBuffer** describes the Microsoft-extended port attributes for the port. **Note** For Windows Vista, Windows Server 2008, and Windows 7, the Microsoft-extended port attributes field is set to zero.

For Windows XP and Windows Server 2003, the Microsoft-extended port attribute might be set to **USB\_PORTATTR\_NO\_OVERCURRENT\_UI**. This value indicates that user interface will be hidden when an overcurrent occurs on the port.

## Output buffer length

`Parameters.DeviceIoControl.OutputBufferLength` indicates the size, in bytes, of the data returned at `SystemBuffer`.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`. If the hub has not been started or is not functional, the request returns `STATUS_UNSUCCESSFUL`.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows XP, Windows Server 2003, and later.
Header	<code>usbiocrtl.h</code> (include <code>Usbiocrtl.h</code> )

## See also

[USB\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_D RIVERKEY\_NAME IOCTL (usbioclt.h)

Article05/18/2021

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME** I/O control request retrieves the driver registry key name that is associated with the device that is connected to the indicated port.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

The **AssociatedIrp.SystemBuffer** member points to a user-allocated [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#) structure. On input, the **ConnectionIndex** member of this structure contains the number of the port that the device is connected to.

## Input buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#) structure.

## Output buffer

On output, the **DriverKeyName** member of the [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#) structure at **AssociatedIrp.SystemBuffer** contains the driver key name that is associated with the device that is connected to the port that is indicated by **ConnectionIndex**.

The **ActualLength** member indicates the length, in bytes, of the driver key name. The **Parameters.DeviceIoControl.OutputBufferLength** member indicates the size, in bytes, of the entire [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#) structure.

## Output buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#) structure.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`.

If the caller allocates an output buffer that is large enough to hold all of the output data, `IoStatus.Information` will be equal to the value of `ActualLength`. If the output buffer is large enough to hold all of the output data, `IoStatus.Information` will be equal to `sizeof(USB_NODE_CONNECTION_DRIVERKEY_NAME)`.

## Requirements

Header	usbiocrtl.h (include Usbiocrtl.h)
--------	-----------------------------------

## See also

[USB\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION IOCTL (usbioclt.h)

Article05/18/2021

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION** request retrieves information about the indicated USB port and the device that is attached to the port, if there is one.

Client drivers must send this IOCTL at an IRQL of PASSIVE\_LEVEL.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

The **AssociatedIrp.SystemBuffer** member points to a user-allocated [USB\\_NODE\\_CONNECTION\\_INFORMATION](#) structure that describes the connection. On input, the **ConnectionIndex** member of this structure contains the port number.

## Input buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_INFORMATION](#) structure.

## Output buffer

On output, the **Parameters.DeviceIoControl.OutputBufferLength** member contains the size of the output data. This size is variable, because it depends on the number of pipes that are associated with the port.

**AssociatedIrp.SystemBuffer** points to a user-allocated [USB\\_NODE\\_CONNECTION\\_INFORMATION](#) structure that contains the output data.

## Output buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_INFORMATION](#) structure.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`.

## Remarks

The `IOCTL_USB_GET_NODE_CONNECTION_INFORMATION_EX` request is an extended version of `IOCTL_USB_GET_NODE_CONNECTION_INFORMATION`. The two requests are identical, except that the extended version of the request can report low, full, and high speed connections and the older `IOCTL_USB_GET_NODE_CONNECTION_INFORMATION` request reports only low and full speed connections. For more information about the difference between these two requests, see [USB\\_NODE\\_CONNECTION\\_INFORMATION](#) and [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#).

## Requirements

Header	usbiocrtl.h (include Usbiocrtl.h)
--------	-----------------------------------

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX IOCTL (usbioclt.h)

Article 02/22/2024

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX** request retrieves information about a USB port and the device that is attached to the port, if there is one.

Client drivers must send this IOCTL at an IRQL of PASSIVE\_LEVEL.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input / Output buffer

Both input and output buffers point to a caller-allocated

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure. On input, the **ConnectionIndex** member of this structure must contain a number greater than or equal to 1 that indicates the number of the port whose connection information is to be reported. The hub driver returns connection information in the remaining members of [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#). The IRP, the **AssociatedIrp.SystemBuffer** member points to the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure.

On output, the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure receives information about the indicated connection from the USB hub driver.

## Input / Output buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure.

## Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

## Remarks

Here is an example that shows how to get the speed of the hub port by sending this request.

The function specifies the symbolic link to hub device and port index to query. On return, pPortSpeed indicates:

- Port speed
- SPEED\_PATHERROR: if unable to open path.
- SPEED\_IOCTLERROR: Hub IOCTL failed.

```
void GetPortSpeed(const WCHAR *Path, ULONG PortIndex, UCHAR *pPortSpeed)

{
    if (Path == NULL) { return; }

    HANDLE handle = CreateFile(
        Path,
        GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_WRITE | FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        NULL,
        NULL
    );

    if (handle == INVALID_HANDLE_VALUE)
    {
        *pPortSpeed = SPEED_PATHERROR;
        return;
    }

    PUSB_NODE_CONNECTION_INFORMATION_EX ConnectionInfo =
        (PUSB_NODE_CONNECTION_INFORMATION_EX)
        malloc(sizeof(USB_NODE_CONNECTION_INFORMATION_EX));

    ConnectionInfo->ConnectionIndex = PortIndex;

    ULONG bytes = 0;
    BOOL success = DeviceIoControl(handle,
        IOCTL_USB_GET_NODE_CONNECTION_INFORMATION_EX,
        ConnectionInfo,
        sizeof(USB_NODE_CONNECTION_INFORMATION_EX),
        ConnectionInfo,
        sizeof(USB_NODE_CONNECTION_INFORMATION_EX),
        &bytes,
        NULL);
```

```
CloseHandle(handle);

if (success == FALSE)
{
    *pPortSpeed = SPEED_IOCTLERROR;
}
else
{
    *pPortSpeed = (UCHAR)ConnectionInfo->Speed;
}

free(ConnectionInfo);
}
```

## Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP, Windows Server 2003, and later.
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB\\_NODE\\_CONNECTION\\_INFORMATION](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 IOCTL (usbioclt.h)

Article05/18/2021

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2** I/O control is sent by an application to retrieve information about the protocols that are supported by a particular USB port on a hub. The request also retrieves the speed capability of the port.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2** is a user-mode I/O control request. This request targets the Universal Serial Bus (USB) hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

**AssociatedIrp.SystemBuffer** points to a caller-allocated [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) structure.

On input, the caller must set the structure members as follows:

- The caller must specify the port number in the **ConnectionIndex** member. **ConnectionIndex** must be a value in the range of 1 to *n*, where *n* is the highest port number retrieved in a previous [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request.
- The caller must set a protocol flag in the **SupportedUsbProtocols** member (see [USB\\_PROTOCOLS](#)). In Windows 8, **SupportedUsbProtocols.Usb300** must be set to 1. Otherwise, the request fails with the **STATUS\_INVALID\_PARAMETER** error code.
- The caller must set the **Length** member to the size, in bytes, of the caller-allocated buffer pointed to by **AssociatedIrp.SystemBuffer**. The size of the buffer must be `sizeof (USB_NODE_CONNECTION_INFORMATION_EX_V2)`.

## Input buffer length

The **Parameters.DeviceIoControl.InputBufferLength** member indicates the size, in bytes, of the caller-allocated buffer whose size equals `sizeof(USB_NODE_CONNECTION_INFORMATION_EX_V2)`.

## Output buffer

On output, the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) structure that is pointed to by **AssociatedIrp.SystemBuffer** is filled with information about the attached device.

## Output buffer length

The **Parameters.DeviceIoControl.OutputBufferLength** member indicates the size, in bytes, of the output buffer **SystemBuffer**.

## Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB driver stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

## Remarks

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_INFORMATION\_EX\_V2** request queries the specified port's hub to get information about protocols supported by the port and the operating speed capability, if a device is attached to the port.

If the request completes successfully, the **SupportedUsbProtocols.Usb200** member of [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) indicates protocols supported by the port. For instance, if the port supports signaling protocol defined by the USB 2.0 specification, then **SupportedUsbProtocols.Usb200** is set to 1.

Note that **SupportedUsbProtocols.Usb110** is always set to 1 for high-speed and full-speed hubs. That is because a high speed-capable hub supports USB 1.1 protocol through split transactions and transaction translators. **SupportedUsbProtocols.Usb110** is never set to 1 for a USB 3.0 port.

In addition, the request also determines whether the port and the attached device are capable of operating at SuperSpeed. If they are, the **Flags.DeviceIsSuperSpeedCapableOrHigher** member is set to 1. If the device attached to the port is currently operating at SuperSpeed, then **DeviceIsOperatingAtSuperSpeedOrHigher** is set to 1.

## Requirements

---

<b>Minimum supported client</b>	Windows 8
<b>Minimum supported server</b>	None supported
<b>Header</b>	usbiocctl.h (include Usbiocctl.h)

# IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME IOCTL (usbioctl.h)

Article05/18/2021

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME** I/O control request is used with the [USB\\_NODE\\_CONNECTION\\_NAME](#) structure to retrieve the symbolic link name of the hub that is attached to the downstream port.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_NAME** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

The **AssociatedIrp.SystemBuffer** member points to a [USB\\_NODE\\_CONNECTION\\_NAME](#) structure. On input, the **ConnectionIndex** member of this structure must indicate the number of the port to check for an attached hub.

## Input buffer length

The size of a [USB\\_NODE\\_CONNECTION\\_NAME](#) structure.

## Output buffer

**AssociatedIrp.SystemBuffer** points to a [USB\\_NODE\\_CONNECTION\\_NAME](#) structure. On output, this structure contains the symbolic name of the attached hub in the **HubName** member. If no hub is attached, the hub does not have a symbolic link, or the attached device is not a hub, **HubName[0]** will contain a value of **UNICODE\_NULL**.

## Output buffer length

The **Parameters.DeviceIoControl.OutputBufferLength** member contains the size, in bytes, of the entire [USB\\_NODE\\_CONNECTION\\_NAME](#) structure.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful.

The request reports success, even if no hub is attached, the attached hub has no symbolic link, or the attached device is not a hub.

Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`.

## Requirements

Header	usbioclt.h (include Usbioclt.h)
--------	---------------------------------

## See also

[USB\\_NODE\\_CONNECTION\\_NAME](#)

# **IOCTL\_USB\_GET\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION IOCTL (usbiioctl.h)**

Article05/22/2024

The **IOCTL\_USB\_GET\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** request retrieves USB port super-speed lane information.

Client drivers must send this IOCTL at an IRQL of PASSIVE\_LEVEL.

**IOCTL\_USB\_GET\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input / Output buffer**

Both input and output buffers point to a caller-allocated **USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** structure. On input, the ConnectionIndex member of this structure must contain a number greater than or equal to 1 that indicates the number of the port whose super-speed lane information is to be reported. The hub driver returns super-speed lane information in the remaining members of the **USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** structure. The IRP, the *AssociatedIrp.SystemBuffer* member points to the **USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** structure.

On output, the **USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** structure receives information about the indicated super-speed lanes from the USB hub driver.

## **Input / Output buffer length**

The size of a **USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION** structure.

## **Status block**

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`.

## Remarks

Here is an example that shows how to retrieve the USB port super-speed lane information.

C++

```
success = DeviceIoControl(hHubDevice,
    IOCTL_USB_GET_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION,
        connectionSSPInfo,
    sizeof(USB_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION),
        connectionSSPInfo,
    sizeof(USB_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION),
        &nBytes,
        NULL);
```

## Requirements

[+] Expand table

Requirement	Value
Header	usbioclt.h

## See also

- [USB\\_NODE\\_CONNECTION\\_SUPERSPEEDPLUS\\_INFORMATION](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# IOCTL\_USB\_GET\_NODE\_INFORMATION

## IOCTL (usbioclt.h)

Article 02/22/2024

The **IOCTL\_USB\_GET\_NODE\_INFORMATION** I/O control request is used with the [USB\\_NODE\\_INFORMATION](#) structure to retrieve information about a parent device.

**IOCTL\_USB\_GET\_NODE\_INFORMATION** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Input buffer

On input, the **AssociatedIrp.SystemBuffer** member points to a [USB\\_NODE\\_INFORMATION](#) structure. On input, the **NodeType** member of this structure must indicate whether the parent device is a hub or a non-hub composite device.

### Input buffer length

The size of a [USB\\_NODE\\_INFORMATION](#) structure.

### Output buffer

On output, **AssociatedIrp.SystemBuffer** points to a [USB\\_NODE\\_INFORMATION](#) structure that holds information about the parent device.

### Output buffer length

The size of a [USB\\_NODE\\_INFORMATION](#) structure.

### Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

# Requirements

 [Expand table](#)

Requirement	Value
Header	usbiocctl.h (include Usbiocctl.h)

## See also

[USB\\_NODE\\_INFORMATION](#)

# IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES IOCTL (usbioctl.h)

Article05/18/2021

The **IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES** I/O control request is sent by an application to retrieve information about a specific port on a USB hub.

**IOCTL\_USB\_GET\_PORT\_CONNECTOR\_PROPERTIES** is a user-mode I/O control request. This request targets the Universal Serial Bus (USB) hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input buffer

`AssociatedIrp.SystemBuffer` points to a caller-allocated [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#) structure. The caller must set the structure members as follows:

- The caller must specify the port number of the port being queried in the `ConnectionIndex` member. `ConnectionIndex` must be a value in the range of 1 to *n*, where *n* is the highest port number retrieved in a previous [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request.
- The caller must specify the `CompanionIndex` member. For a SuperSpeed hub, the caller must set the `CompanionIndex` member to 0.

The caller can use `CompanionIndex` to get port numbers associated with the port being queried. If more than one companion port is associated with the port, to get all port numbers, the application can send the request in a loop. Start with `CompanionIndex` set to 0 and increment the `CompanionIndex` value in each iteration until the request completes with `CompanionPortNumber` set to 0 and `CompanionHubSymbolicLinkName` is NULL.

## Input buffer length

The `Parameters.DeviceIoControl.InputBufferLength` member indicates the size, in bytes, of the caller-allocated buffer pointed to by `AssociatedIrp.SystemBuffer`. For more information see Remarks.

## Output buffer

On output, [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#), pointed to by `AssociatedIrp.SystemBuffer`, is filled with information about the physical connector associated with the port.

## Output buffer length

The `Parameters.DeviceIoControl.OutputBufferLength` member indicates the size, in bytes, of the output buffer `SystemBuffer`.

## Status block

The USB stack sets `Irp->IoStatus.Status` to `STATUS_SUCCESS` if the request is successful. Otherwise, the USB stack sets `Status` to the appropriate error condition, such as `STATUS_INVALID_PARAMETER` or `STATUS_INSUFFICIENT_RESOURCES`. If the hub has not been started or is not functional, the request returns `STATUS_UNSUCCESSFUL`.

## Remarks

The caller must supply a buffer that is large enough to hold a [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#) structure and the symbolic link name of a companion hub, if such a hub is associated with the specified port. To obtain the size of the buffer to allocate, the caller must send a `IOCTL_USB_GET_PORT_CONNECTOR_PROPERTIES` request. In the request, `AssociatedIrp.SystemBuffer` must be a caller-allocated `USB_PORT_CONNECTOR_PROPERTIES` structure and `Parameters.DeviceIoControl.InputBufferLength` must be `sizeof(USB_PORT_CONNECTOR_PROPERTIES)`. Upon successful completion, the `ActualLength` member of `USB_PORT_CONNECTOR_PROPERTIES` indicates the actual size of the buffer. If a symbolic link name exists, the received value includes the size of the string that stores the link name.

Based on the `ActualLength` value, the caller can then allocate a buffer and send the `IOCTL_USB_GET_PORT_CONNECTOR_PROPERTIES` request again. After the request completes, the buffer is filled with port information and the `CompanionHubSymbolicLinkName` member is a Unicode string that contains the symbolic link name of the companion hub.

## Requirements

Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbiocrtl.h (include Usbiocrtl.h)

## See also

[IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

[USB\\_HUB\\_INFORMATION\\_EX](#)

[USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

# IOCTL\_USB\_GET\_ROOT\_HUB\_NAME

## IOCTL (usbioclt.h)

Article02/22/2024

The **IOCTL\_USB\_GET\_ROOT\_HUB\_NAME** I/O control request is used with the [USB\\_ROOT\\_HUB\\_NAME](#) structure to retrieve the symbolic link name of the root hub.

**IOCTL\_USB\_GET\_ROOT\_HUB\_NAME** is a user-mode I/O control request. This request targets the USB host controller (GUID\_DEVINTERFACE\_USB\_HOST\_CONTROLLER).

### Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

### Input buffer

None.

### Input buffer length

None.

### Output buffer

On output, the **AssociatedIrp.SystemBuffer** member points to a [USB\\_ROOT\\_HUB\\_NAME](#) structure that contains the symbolic link name of the root hub. The leading "\xxx\ " text is not included in the retrieved string.

### Output buffer length

The size of a [USB\\_ROOT\\_HUB\\_NAME](#) structure.

### Status block

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

If the root hub is removed or stopped, the request returns STATUS\_SUCCESS but the string is NULL.

## Requirements

 Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB\\_ROOT\\_HUB\\_NAME](#)

# IOCTL\_USB\_GET\_TRANSPORT\_CHARACTERISTICS IOCTL (usbioclt.h)

Article02/22/2024

The client driver sends this request to retrieve the transport characteristics.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input / Output buffer

The **AssociatedIrp.SystemBuffer** member is a pointer to a caller-allocated [USB\\_TRANSPORT\\_CHARACTERISTICS](#) structure. On input, set **Version** to [USB\\_TRANSPORT\\_CHARACTERISTICS\\_VERSION\\_1](#). On output **Version** is reset to a version less than or equal to [USB\\_TRANSPORT\\_CHARACTERISTICS\\_VERSION\\_1](#); the remaining members of the structure is filled with transport information.

## Input / Output buffer length

The size of the [USB\\_TRANSPORT\\_CHARACTERISTICS](#) structure.

## Status block

**Irp->IoStatus.Status** is set to [STATUS\\_SUCCESS](#) if the request is successful. Otherwise, **Status** to the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

This request retrieves the transport characteristics to decide on an algorithm for streaming. For example, a display driver can use the latency and bandwidth information to decide its codec selection.

This information might not be always available. The USB driver stack depends on the underlying transport to expose these values. Therefore, the client driver must have a back up mechanism for such cases where the request.

If the client driver is interested in knowing the latest information at all times, the driver must register for notification when transport characteristics change, keep a request

pending with the USB driver stack, and unregister when the notification is no longer required. The driver can accomplish all those tasks by sending these IOCTL requests.

- [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)
- [IOCTL\\_USB\\_NOTIFY\\_ON\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)
- [IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioctl.h
IRQL	<=DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

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

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# IOCTL\_USB\_HCD\_DISABLE\_PORT IOCTL (usbioc.h)

Article02/22/2024

The **IOCTL\_USB\_HCD\_DISABLE\_PORT** IOCTL has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Available on Microsoft Windows Server 2003, Windows XP, and Windows 2000, but it is not available on Windows Vista.
Header	usbioc.h (include Usbioc.h)

# IOCTL\_USB\_HCD\_ENABLE\_PORT IOCTL (usbioc.h)

Article02/22/2024

The **IOCTL\_USB\_HCD\_ENABLE\_PORT** IOCTL has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Available on Microsoft Windows Server 2003, Windows XP, and Windows 2000, but it is not available on Windows Vista.
Header	usbioc.h (include Usbioc.h)

# IOCTL\_USB\_HCD\_GET\_STATS\_1 IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_USB\_HCD\_GET\_STATS\_1** IOCTL has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Available on Microsoft Windows Server 2003, Windows XP, and Windows 2000, but it is not available on Windows Vista.
Header	usioctl.h (include Usioctl.h)

# IOCTL\_USB\_HCD\_GET\_STATS\_2 IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_USB\_HCD\_GET\_STATS\_2** IOCTL has been deprecated. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Available on Microsoft Windows Server 2003, Windows XP, and Windows 2000, but it is not available on Windows Vista.
Header	usioctl.h (include Usioctl.h)

# **IOCTL\_USB\_HUB\_CYCLE\_PORT IOCTL (usbioclt.h)**

Article02/22/2024

The **IOCTL\_USB\_HUB\_CYCLE\_PORT** I/O control request power-cycles the port that is associated with the PDO that receives the request.

**IOCTL\_USB\_HUB\_CYCLE\_PORT** is a user-mode I/O control request. This request targets the USB hub device (GUID\_DEVINTERFACE\_USB\_HUB).

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

The **AssociatedIrp.SystemBuffer** member points to a caller-allocated [USB\\_CYCLE\\_PORT\\_PARAMS](#) structure that specifies the port number.

## **Input buffer length**

The size of a [USB\\_CYCLE\\_PORT\\_PARAMS](#) structure.

## **Output buffer**

None.

## **Output buffer length**

None.

## **Status block**

The USB stack sets **Irp->IoStatus.Status** to **STATUS\_SUCCESS** if the request is successful. Otherwise, the USB stack sets **Status** to the appropriate error condition, such as **STATUS\_INVALID\_PARAMETER** or **STATUS\_INSUFFICIENT\_RESOURCES**.

## **Remarks**

You can also power cycle the port by using the **Device Manager's Enable/Disable** feature. This feature causes the bus driver to reset the device. Alternatively, you can use DevCon to enable or disable the device.

The executable for DevCon can be found in the

`<install_path>\WinDDK\build_number\tools\devcon\<arch>\devcon.exe` folder.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Supported on Windows 8 and later versions of Windows, if the caller is running as Administrator. Supported on Microsoft Windows Server 2003, Windows XP-based versions of Windows. Not supported on Windows 7, Windows Vista, and Windows Server 2008.
Header	usbiocrtl.h (include Usbiocrtl.h)

# **IOCTL\_USB\_NOTIFY\_ON\_TRANSPORT\_C HARACTERISTICS\_CHANGE IOCTL (usbioclt.h)**

Article05/18/2021

This request notifies the caller of change in transport characteristics.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input / Output buffer**

The [AssociatedIrp.SystemBuffer](#) member is a pointer to a caller-allocated [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#) structure. On input the caller passes the registration handle retrieved in the previous [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request. On output, the structure is filled with the latest information about the type of information for which the caller.

## **Input / Output buffer length**

The size of the [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_NOTIFICATION](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

This request is kept pending by the USB driver stack until there is a change in the transport characteristics for which the caller registered. On completion of this request, the USB driver stack returns the information in the output buffer.

## **Requirements**

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioc.h
IRQL	<=DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

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

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# IOCTL\_USB\_REGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE IOCTL (usbioclt.h)

Article05/18/2021

This request registers for notifications about the changes in transport characteristics.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Input / Output buffer

The `AssociatedIrp.SystemBuffer` member is a pointer to a [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_REGISTRATION](#) structure. On input, the client driver can specify the type of notification changes in which the driver is interested by setting the flags in the `ChangeNotificationInputFlags` member.

On output, the structure is filled with the registration handle and initial values of the transport characteristics.

## Input / Output buffer length

The size of the [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_REGISTRATION](#) structure.

## Status block

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates the appropriate error condition as a [NTSTATUS](#) code.

## Remarks

The transport characteristics of MA-USB mediums can vary significantly over time. If the client driver is interested in knowing the latest information at all times, the driver must register for notification by sending the request.

This request can be sent by a user mode application, UMDF driver, or a KMDF driver. USB driver stack checks for stale and bad registration handle. If the request is received

on a handle before registration and after unregistration, the driver stack fails the request.

## Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioclt.h
IRQL	<=DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

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

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# IOCTL\_USB\_RESET\_HUB IOCTL (usioctl.h)

Article02/22/2024

The **IOCTL\_USB\_RESET\_HUB** IOCTL is used by the USB driver stack. Do not use.

## Major code

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## Status block

Irp->IoStatus.Status is set to STATUS\_SUCCESS if the request is successful.

Otherwise, Status to the appropriate error condition as a NTSTATUS code.

For more information, see [NTSTATUS Values](#).

## Requirements

  [Expand table](#)

Requirement	Value
Header	usioctl.h

# **IOCTL\_USB\_START\_TRACKING\_FOR\_TIME\_SYNC IOCTL (usbioclt.h)**

Article03/29/2024

This request registers the caller with USB driver stack for time sync services.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input / Output buffer**

A pointer to a [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure. On input, the caller must set the **TimeTrackingHandle** member to NULL. On output, the USB driver stack sets the **TimeTrackingHandle** member to a handle that tracks the sync services operation.

## **Input / Output buffer length**

The size of the [USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates an appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

When this IOCTL request completes, the USB driver stack enables certain interrupts from the host controller to keep track of closest frame/microframe boundary in order to predict the system QPC value with accuracy. Enabling the hardware interrupts adds an overhead to the power consumption because the CPU wakes up every 2.048 seconds when working in the D0 power state. Therefore we recommend that the caller should register for time sync services only when needed.

The driver stack disables those interrupts when it receives and completes the [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioclt.h
IRQL	<= DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# **IOCTL\_USB\_STOP\_TRACKING\_FOR\_TIME\_SYNC IOCTL (usbioc.h)**

Article02/22/2024

This request unregisters the caller with USB driver stack for time sync services.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure that contains the time tracking handle previously received through the [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

## **Input buffer length**

The size of the [USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC\\_INFORMATION](#) structure.

## **Status block**

`Irп->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates an appropriate error condition as a [NTSTATUS](#) code.

## **Requirements**

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioc.h
IRQL	<= DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# **IOCTL\_USB\_UNREGISTER\_FOR\_TRANSPORT\_CHARACTERISTICS\_CHANGE IOCTL (usbioclt.h)**

Article05/18/2021

This request unregisters the caller from getting notifications about transport characteristics changes.

## **Major code**

[IRP\\_MJ\\_DEVICE\\_CONTROL](#)

## **Input buffer**

A pointer to a [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_UNREGISTRATION](#) structure that contains the registration handle previously received by the [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request.

## **Input buffer length**

The size of the [USB\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE\\_UNREGISTRATION](#) structure.

## **Status block**

`Irp->IoStatus.Status` is set to `STATUS_SUCCESS` if the request is successful. Otherwise, `Status` indicates the appropriate error condition as a [NTSTATUS](#) code.

## **Remarks**

After this request completes the registration handle is considered to be stale and requests sent on that handle are failed by the USB driver stack.

## **Requirements**



<b>Header</b>	usbiocli.h
<b>IRQL</b>	<=DISPATCH_LEVEL

## See also

[Creating IOCTL Requests in Drivers](#)

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

[WdfIoTargetSendInternalIoctlOthersSynchronously](#)

[WdfIoTargetSendInternalIoctlSynchronously](#)

[WdfIoTargetSendIoctlSynchronously](#)

# USB\_BUS\_NOTIFICATION structure (usbioc.h)

Article 02/22/2024

Stores certain bus information. This structure is used in the [IOCTL\\_INTERNAL\\_USB\\_GET\\_BUS\\_INFO](#) request.

## Syntax

C++

```
typedef struct _USB_BUS_NOTIFICATION {
    USB_NOTIFICATION_TYPE NotificationType;
    ULONG             TotalBandwidth;
    ULONG             ConsumedBandwidth;
    ULONG             ControllerNameLength;
} USB_BUS_NOTIFICATION, *PUSB_BUS_NOTIFICATION;
```

## Members

`NotificationType`

A [USB\\_NOTIFICATION\\_TYPE](#)-value that indicates the type of notification.

`TotalBandwidth`

The total bandwidth, in bits per second, available on the bus.

`ConsumedBandwidth`

The mean bandwidth already in use, in bits per second.

`ControllerNameLength`

The length of the Unicode symbolic name (in bytes) for the host controller to which this device is attached. The length does not include NULL.

## Requirements

[+] [Expand table](#)

<b>Requirement</b>	<b>Value</b>
Header	usioctl.h

# USB\_CONNECTION\_STATUS enumeration (usbioc.h)

Article 06/03/2021

The **USB\_CONNECTION\_STATUS** enumerator indicates the status of the connection to a device on a USB hub port.

## Syntax

C++

```
typedef enum _USB_CONNECTION_STATUS {
    NoDeviceConnected,
    DeviceConnected,
    DeviceFailedEnumeration,
    DeviceGeneralFailure,
    DeviceCausedOvercurrent,
    DeviceNotEnoughPower,
    DeviceNotEnoughBandwidth,
    DeviceHubNestedTooDeeply,
    DeviceInLegacyHub,
    DeviceEnumerating,
    DeviceReset
} USB_CONNECTION_STATUS, *PUSB_CONNECTION_STATUS;
```

## Constants

NoDeviceConnected

Indicates that there is no device connected to the port.

DeviceConnected

Indicates that a device was successfully connected to the port.

DeviceFailedEnumeration

Indicates that an attempt was made to connect a device to the port, but the enumeration of the device failed.

DeviceGeneralFailure

Indicates that an attempt was made to connect a device to the port, but the connection failed for unspecified reasons.

**DeviceCausedOvercurrent**

Indicates that an attempt was made to connect a device to the port, but the attempt failed because of an overcurrent condition.

**DeviceNotEnoughPower**

Indicates that an attempt was made to connect a device to the port, but there was not enough power to drive the device, and the connection failed.

**DeviceNotEnoughBandwidth**

Indicates that an attempt was made to connect a device to the port, but there was not enough bandwidth available for the device to function properly, and the connection failed.

**DeviceHubNestedTooDeeply**

Indicates that an attempt was made to connect a device to the port, but the nesting of USB hubs was too deep, so the connection failed.

**DeviceInLegacyHub**

Indicates that an attempt was made to connect a device to the port of an unsupported legacy hub, and the connection failed.

**DeviceEnumerating**

Indicates that a device connected to the port is currently being enumerated.

**Note** This constant is supported in Windows Vista and later operating systems.

**DeviceReset**

Indicates that device connected to the port is currently being reset.

**Note** This constant is supported in Windows Vista and later operating systems.

## Remarks

The USB bus driver reports connection status in a [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure in response to an [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) request.

## Requirements

Header	usbioctl.h (include Usbioctl.h)
--------	---------------------------------

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

[USB Constants and Enumerations](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

# USB\_CYCLE\_PORT\_PARAMS structure (usbioc.h)

Article 02/22/2024

The **USB\_CYCLE\_PORT\_PARAMS** structure is used with the [IOCTL\\_USB\\_HUB\\_CYCLE\\_PORT](#) I/O control request to power cycle the port that is associated with the PDO that receives the request.

## Syntax

C++

```
typedef struct _USB_CYCLE_PORT_PARAMS {
    ULONG ConnectionIndex;
    ULONG StatusReturned;
} USB_CYCLE_PORT_PARAMS, *PUSB_CYCLE_PORT_PARAMS;
```

## Members

`ConnectionIndex`

Specifies the port number starting at 1.

`StatusReturned`

On return, contains the USBD status of the operation.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioc.h

## See also

[IOCTL\\_USB\\_HUB\\_CYCLE\\_PORT](#)

# USB\_DESCRIPTOR\_REQUEST structure (usbioc.h)

Article 02/22/2024

The **USB\_DESCRIPTOR\_REQUEST** structure is used with the [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request to retrieve one or more descriptors for the device that is associated with the indicated connection index. The fields in this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#).

## Syntax

C++

```
typedef struct _USB_DESCRIPTOR_REQUEST {
    ULONG ConnectionIndex;
    struct {
        UCHAR bmRequest;
        UCHAR bRequest;
        USHORT wValue;
        USHORT wIndex;
        USHORT wLength;
    } SetupPacket;
    UCHAR Data[0];
} USB_DESCRIPTOR_REQUEST, *PUSB_DESCRIPTOR_REQUEST;
```

## Members

**ConnectionIndex**

The port whose descriptors are retrieved.

**SetupPacket**

The members of the **SetupPacket** structure defined as per the official specification. See section 9.3.

**SetupPacket.bmRequest**

The type of USB device request (standard, class, or vendor), the direction of the data transfer, and the type of data recipient (device, interface, or endpoint). On input to the [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request, the USB

stack ignores the value of **bmRequest** and inserts a value of 0x80. This value indicates a standard USB device request and a device-to-host data transfer.

#### SetupPacket.bRequest

The request number. On input to the [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request, the USB stack ignores the value of **bRequest** and inserts a value of 0x06. This value indicates a request of **GET\_DESCRIPTOR**.

#### SetupPacket.wValue

On input to the [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request, the caller should specify the type of descriptor to retrieve in the high byte of **wValue** and the descriptor index in the low byte. The following table lists the possible descriptor types. These constant types are defined in the Usbspec.h header included in the Windows Driver Kit.

[+] [Expand table](#)

Descriptor type	Meaning
USB_DEVICE_DESCRIPTOR_TYPE	Instructs the USB stack to return the device descriptor.
USB_CONFIGURATION_DESCRIPTOR_TYPE	Instructs the USB stack to return the configuration descriptor and all interface, endpoint, class-specific, and vendor-specific descriptors associated with the current configuration.
USB_STRING_DESCRIPTOR_TYPE	Instructs the USB stack to return the indicated string descriptor.
USB_INTERFACE_DESCRIPTOR_TYPE	Instructs the USB stack to return the indicated interface descriptor.
USB_ENDPOINT_DESCRIPTOR_TYPE	Instructs the USB stack to return the indicated endpoint descriptor.

#### SetupPacket.wIndex

The device-specific index of the descriptor that is to be retrieved.

#### SetupPacket.wLength

The length of the data that is transferred during the second phase of the control transfer.

## Data[0]

On output from the [IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#) I/O control request, this member contains the retrieved descriptors.

## Remarks

If the caller specifies a value of [USB\\_CONFIGURATION\\_DESCRIPTOR\\_TYPE](#) in the **wValue** member, the output buffer must be large enough to hold all of the descriptors that are associated with the current configuration, or the request will fail.

## Requirements

 [Expand table](#)

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_DESCRIPTOR\\_FROM\\_NODE\\_CONNECTION](#)

[USB Structures](#)

# USB\_DEVICE\_CHARACTERISTICS structure (usbioctl.h)

Article09/01/2022

Contains information about the USB device's characteristics, such as the maximum send and receive delays for any request. This structure is used in the [IOCTL\\_USB\\_GET\\_DEVICE\\_CHARACTERISTICS](#) request.

## Syntax

C++

```
typedef struct _USB_DEVICE_CHARACTERISTICS {
    ULONG Version;
    ULONG Reserved[2];
    ULONG UsbDeviceCharacteristicsFlags;
    ULONG MaximumSendPathDelayInMilliSeconds;
    ULONG MaximumCompletionPathDelayInMilliSeconds;
} USB_DEVICE_CHARACTERISTICS, *PUSB_DEVICE_CHARACTERISTICS;
```

## Members

**Version**

The version should be set to `USB_DEVICE_CHARACTERISTICS_VERSION_1`.

**Reserved[2]**

Reserved.

**UsbDeviceCharacteristicsFlags**

A bitmask of flags that indicates to the client driver the transport characteristics that are available and are returned by this structure.

If the `USB_DEVICE_CHARACTERISTICS_MAXIMUM_PATH_DELAYS_AVAILABLE` flag is set, `MaximumSendPathDelayInMilliSeconds` and `MaximumCompletionPathDelayInMilliSeconds` contain valid information. Otherwise they are not available and must not be used by the client driver.

**MaximumSendPathDelayInMilliSeconds**

Contains the maximum delay in milliseconds for any request that is submitted by the client driver and is received by the USB driver stack to the time it is programmed in the host controller, including the maximum delay associated with the network medium if it is a MA-USB host controller.

#### `MaximumCompletionPathDelayInMilliSeconds`

Contains the maximum delay in milliseconds the host controller completes any request for the device to the time the request is completed and sent back to the client driver. For a MA-USB controller this includes any delay associated with the network medium.

## Requirements

Header	usbioctl.h

## See also

[IOCTL\\_USB\\_GET\\_DEVICE\\_CHARACTERISTICS](#)

# USB\_FRAME\_NUMBER\_AND\_QPC\_FOR\_TIME\_SYNC\_INFORMATION structure (usbioc.h)

Article04/01/2021

Stores the frame and microframe numbers and the calculated system QPC values. This structure is used in the [IOCTL\\_USB\\_GET\\_FRAME\\_NUMBER\\_AND\\_QPC\\_FOR\\_TIME\\_SYNC](#) request.

## Syntax

C++

```
typedef struct _USB_FRAME_NUMBER_AND_QPC_FOR_TIME_SYNC_INFORMATION {
    HANDLE          TimeTrackingHandle;
    ULONG           InputFrameNumber;
    ULONG           InputMicroFrameNumber;
    LARGE_INTEGER   QueryPerformanceCounterAtInputFrameOrMicroFrame;
    LARGE_INTEGER   QueryPerformanceCounterFrequency;
    ULONG           PredictedAccuracyInMicroSeconds;
    ULONG           CurrentGenerationID;
    LARGE_INTEGER   CurrentQueryPerformanceCounter;
    ULONG           CurrentHardwareFrameNumber;
    ULONG           CurrentHardwareMicroFrameNumber;
    ULONG           CurrentUSBFrameNumber;
} USB_FRAME_NUMBER_AND_QPC_FOR_TIME_SYNC_INFORMATION,
*PUSB_FRAME_NUMBER_AND_QPC_FOR_TIME_SYNC_INFORMATION;
```

## Members

TimeTrackingHandle

The time racking handle received in the previous [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

InputFrameNumber

A 32-bit USB bus frame number.

InputMicroFrameNumber

A 3-bit value received from the hardware.

### `QueryPerformanceCounterAtInputFrameOrMicroFrame`

A value predicted by the USB driver stack that represents the system QPC value at the beginning of the frame and microframe represented by the **InputFrameNumber** and **InputMicroFrameNumber** input values.

### `QueryPerformanceCounterFrequency`

The current performance-counter frequency, in counts per second.

### `PredictedAccuracyInMicroSeconds`

A value that represents the accuracy of the predicted QPC value in micro seconds.

### `CurrentGenerationID`

An identifier for this request of time synchronization.

### `CurrentQueryPerformanceCounter`

Current QPC value captured that is synchronized with the bus frame numbers represented by **CurrentHardwareFrameNumber**, **CurrentHardwareMicroFrameNumber** and **CurrentUSBFrameNumber**.

### `CurrentHardwareFrameNumber`

A 1-bit value of the current hardware frame number that is directly read from the MFINDEX register.

### `CurrentHardwareMicroFrameNumber`

A 3-bit value of the current hardware micro frame number that is directly read from the MFINDEX register.

### `CurrentUSBFrameNumber`

A 32-bit USB frame number value returned by [\\_URB\\_GET\\_CURRENT\\_FRAME\\_NUMBER](#).

## Requirements

Header	usbioctl.h
--------	------------

# USB\_HCD\_DRIVERKEY\_NAME structure (usbioc.h)

Article 02/22/2024

The **USB\_HCD\_DRIVERKEY\_NAME** structure is used with the [IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#) I/O control request to retrieve the driver key in the registry for the USB host controller driver.

## Syntax

C++

```
typedef struct _USB_HCD_DRIVERKEY_NAME {
    ULONG ActualLength;
    WCHAR DriverKeyName[1];
} USB_HCD_DRIVERKEY_NAME, *PUSB_HCD_DRIVERKEY_NAME;
```

## Members

**ActualLength**

The length, in bytes, of the string in the **DriverKeyName** member.

**DriverKeyName[1]**

A NULL-terminated Unicode driver key name for the USB host controller.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioc.h (include Usbioc.h)

## See also

[IOCTL\\_GET\\_HCD\\_DRIVERKEY\\_NAME](#)

## USB Structures

# USB\_HUB\_CAP\_FLAGS union (usbioc.h)

Article 04/01/2021

The `USB_HUB_CAP_FLAGS` structure is used to report the capabilities of a hub.

## Syntax

C++

```
typedef union _USB_HUB_CAP_FLAGS {
    ULONG ul;
    struct {
        ULONG HubIsHighSpeedCapable : 1;
        ULONG HubIsHighSpeed : 1;
        ULONG HubIsMultiTtCapable : 1;
        ULONG HubIsMultiTt : 1;
        ULONG HubIsRoot : 1;
        ULONG HubIsArmedWakeOnConnect : 1;
        ULONG HubIsBusPowered : 1;
        ULONG ReservedMBZ : 25;
    };
} USB_HUB_CAP_FLAGS, *PUSB_HUB_CAP_FLAGS;
```

## Members

`ul`

A bitmask that represents the hub capabilities.

`HubIsHighSpeedCapable`

If **TRUE**, the hub is high speed-capable. This capability does not necessarily mean that the hub is operating at high speed

`HubIsHighSpeed`

If **TRUE**, the hub is high speed.

`HubIsMultiTtCapable`

If **TRUE**, the hub is capable of doing multiple transaction translations simultaneously.

`HubIsMultiTt`

If **TRUE**, the hub is configured to perform multiple transaction translations simultaneously.

`HubIsRoot`

If **TRUE**, the hub is the root hub.

`HubIsArmedWakeOnConnect`

If **TRUE**, the hub is armed to wake when a device is connected to the hub.

`HubIsBusPowered`

A boolean value that indicates whether the hub is bus-powered. **TRUE**, the hub is bus-powered; **FALSE**, the hub is self-powered.

`ReservedMBZ`

Reserved. Do not use.

## Requirements

<b>Header</b>	<code>usbioctl.h</code> (include <code>Usbioctl.h</code> )

## See also

[USB Structures](#)

[USB\\_HUB\\_CAPABILITIES\\_EX](#)

# USB\_HUB\_CAPABILITIES structure (usbioc.h)

Article02/22/2024

The **USB\_HUB\_CAPABILITIES** structure has been deprecated. Use [USB\\_HUB\\_CAPABILITIES\\_EX](#) instead.

## Syntax

C++

```
typedef struct _USB_HUB_CAPABILITIES {
    ULONG HubIs2xCapable : 1;
} USB_HUB_CAPABILITIES, *PUSB_HUB_CAPABILITIES;
```

## Members

`HubIs2xCapable`

If **TRUE**, the hub is capable of running at high speed.

## Requirements

  [Expand table](#)

Requirement	Value
Header	usbioc.h (include Usbioc.h)

## See also

[IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES](#)

[USB Structures](#)

[USB\\_HUB\\_CAPABILITIES\\_EX](#)

[USB\\_HUB\\_CAP\\_FLAGS](#)

# USB\_HUB\_CAPABILITIES\_EX structure (usbioc.h)

Article02/22/2024

The **USB\_HUB\_CAPABILITIES\_EX** structure is used with the [IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES](#) I/O control request to retrieve the capabilities of a particular USB hub.

## Syntax

C++

```
typedef struct _USB_HUB_CAPABILITIES_EX {
    USB_HUB_CAP_FLAGS CapabilityFlags;
} USB_HUB_CAPABILITIES_EX, *PUSB_HUB_CAPABILITIES_EX;
```

## Members

CapabilityFlags

A [USB\\_HUB\\_CAP\\_FLAGS](#) structure that reports the hub capabilities.

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Available in Windows Vista and later operating systems.
Header	usbioc.h (include Usbioc.h)

## See also

[IOCTL\\_USB\\_GET\\_HUB\\_CAPABILITIES](#)

[USB Structures](#)

[USB\\_HUB\\_CAP\\_FLAGS](#)

# USB\_HUB\_INFORMATION structure (usbioclt.h)

Article 02/22/2024

The **USB\_HUB\_INFORMATION** structure contains information about a hub.

## Syntax

C++

```
typedef struct _USB_HUB_INFORMATION {
    USB_HUB_DESCRIPTOR HubDescriptor;
    BOOLEAN            HubIsBusPowered;
} USB_HUB_INFORMATION, *PUSB_HUB_INFORMATION;
```

## Members

HubDescriptor

A [USB\\_HUB\\_DESCRIPTOR](#) structure that contains selected information from the hub descriptor.

HubIsBusPowered

A Boolean value that indicates whether the hub is bus-powered. **TRUE**, the hub is bus-powered; **FALSE**, the hub is self-powered.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioclt.h (include Usbioclt.h)

## See also

[USB Structures](#)

USB\_HUB\_DESCRIPTOR

USB\_NODE\_INFORMATION

# USB\_HUB\_INFORMATION\_EX structure (usbioc.h)

Article04/01/2021

The **USB\_HUB\_INFORMATION\_EX** structure is used with the [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request to retrieve information about a Universal Serial Bus (USB) hub.

## Syntax

C++

```
typedef struct _USB_HUB_INFORMATION_EX {
    USB_HUB_TYPE HubType;
    USHORT HighestPortNumber;
    union {
        USB_HUB_DESCRIPTOR     UsbHubDescriptor;
        USB_30_HUB_DESCRIPTOR Usb30HubDescriptor;
    } u;
} USB_HUB_INFORMATION_EX, *PUSB_HUB_INFORMATION_EX;
```

## Members

**HubType**

The type of hub: root hub, USB 2.0, or USB 3.0 hub. On successful completion of the [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request, **HubType** contains a [USB\\_HUB\\_TYPE](#) enumerator that indicates the type of hub.

**HighestPortNumber**

Indicates the number of ports on the hub. The ports are numbered from 1 to **HighestPortNumber**, where **HighestPortNumber** is the highest valid port number on the hub.

**u**

**u.UsbHubDescriptor**

If **HubType** indicates a USB 2.0 hub, **u.UsbHubDescriptor** is a [USB\\_HUB\\_DESCRIPTOR](#) structure that contains selected information from the USB 2.0/1.1 hub descriptor, as

defined in the USB 2.0 Specification.

#### u.Usb30HubDescriptor

If **HubType** indicates a USB 3.0 hub, **u.UsbHub30Descriptor** is a [USB\\_30\\_HUB\\_DESCRIPTOR](#) structure that contains selected information from the USB 3.0 hub descriptor, as defined in the USB 3.0 Specification.

## Requirements

Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#)

[USB\\_30\\_HUB\\_DESCRIPTOR](#)

[USB\\_HUB\\_DESCRIPTOR](#)

[USB\\_HUB\\_TYPE](#)

# USB\_HUB\_NAME structure (usbioc.h)

Article 02/22/2024

The **USB\_HUB\_NAME** structure stores the hub's symbolic device name.

## Syntax

C++

```
typedef struct _USB_HUB_NAME {
    ULONG ActualLength;
    WCHAR HubName[1];
} USB_HUB_NAME, *PUSB_HUB_NAME;
```

## Members

ActualLength

The size of the Unicode string pointed to by **HubName**. The **ActualLength** value indicates the length of the string and not the entire structure.

HubName[1]

A NULL-terminated Unicode string that contains the hub's symbolic device name.

## Requirements

[] Expand table

Requirement	Value
Header	usbioc.h (include Usbioc.h)

## See also

[IOCTL\\_INTERNAL\\_USB\\_GET\\_CONTROLLER\\_NAME](#)

[USB Structures](#)

# USB\_HUB\_NODE enumeration (usbiioctl.h)

Article02/22/2024

The **USB\_HUB\_NODE** enumerator indicates whether a device is a hub or a composite device.

## Syntax

C++

```
typedef enum _USB_HUB_NODE {
    UsbHub,
    UsbMIParent
} USB_HUB_NODE;
```

## Constants

[ ] Expand table

**UsbHub**

Indicates that the device is a hub.

**UsbMIParent**

Indicates that the device is a composite device with multiple interfaces.

## Remarks

Composite devices are devices that have multiple interfaces. Windows loads the USB generic parent driver for composite devices, instead of the hub driver, but the generic parent driver performs many of the functions of the hub driver. It creates a child PDO for each interface, as though the interface were a separate device.

## Requirements

[ ] Expand table

Requirement	Value
Header	usbiocctl.h (include Usbiocctl.h)

## See also

[USB Constants and Enumerations](#)

[USB\\_NODE\\_INFORMATION](#)

# USB\_HUB\_TYPE enumeration (usbioclt.h)

Article02/22/2024

The **USB\_HUB\_TYPE** enumeration defines constants that indicate the type of USB hub.

The hub type is retrieved by the [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request.

The request retrieves the hub descriptor associated with the specified hub in the [USB\\_HUB\\_INFORMATION\\_EX](#) structure. The **HubType** member contains a **USB\_HUB\_TYPE** enumerator that the application can use to evaluate the type of hub descriptor retrieved by the request.

## Syntax

C++

```
typedef enum _USB_HUB_TYPE {
    UsbRootHub = 1,
    Usb20Hub = 2,
    Usb30Hub = 3
} USB_HUB_TYPE;
```

## Constants

[+] [Expand table](#)

**UsbRootHub**

Indicates a root hub.

**Usb20Hub**

Indicates that the retrieved hub descriptor is defined in USB 2.0 and 1.1 specifications.

**Usb30Hub**

Indicates that the retrieved hub descriptor is defined in USB 3.0 specification.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbiocnl.h (include Usbiocnl.h)

## See also

[IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#)

[USB Constants and Enumerations](#)

[USB\\_HUB\\_INFORMATION\\_EX](#)

# USB\_ID\_STRING structure (usbioc.h)

Article 02/22/2024

The **USB\_ID\_STRING** structure is used to store a string or multi-string.

## Syntax

C++

```
typedef struct _USB_ID_STRING {
    USHORT LanguageId;
    USHORT Pad;
    ULONG LengthInBytes;
    PWCHAR Buffer;
} USB_ID_STRING, *PUSB_ID_STRING;
```

## Members

LanguageId

Indicates that language ID of the string.

Pad

LengthInBytes

Indicates the length (in bytes) of the string pointed to by **Buffer**, including the terminating **NULL**.

Buffer

Pointer to a string or multi-string.

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Available in Windows Vista and later operating systems.
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB Structures](#)

# USB\_MI\_PARENT\_INFORMATION structure (usbioctl.h)

Article02/22/2024

The **USB\_MI\_PARENT\_INFORMATION** structure contains information about a composite device.

## Syntax

C++

```
typedef struct _USB_MI_PARENT_INFORMATION {
    ULONG NumberOfInterfaces;
} USB_MI_PARENT_INFORMATION, *PUSB_MI_PARENT_INFORMATION;
```

## Members

NumberOfInterfaces

The number of interfaces on the composite device.

## Remarks

A composite device is a device with multiple interfaces (MI). The USB stack treats the interfaces of a composite device as child devices of the composite device and creates a separate PDO for each interface.

## Requirements

[+] Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[USB Structures](#)

## **USB\_NODE\_INFORMATION**

# USB\_NODE\_CONNECTION\_ATTRIBUTES structure (usbioctl.h)

Article02/22/2024

The **USB\_NODE\_CONNECTION\_ATTRIBUTES** structure is used with the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) I/O control request to retrieve the attributes of a connection.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_ATTRIBUTES {
    ULONG             ConnectionIndex;
    USB_CONNECTION_STATUS ConnectionStatus;
    ULONG             PortAttributes;
} USB_NODE_CONNECTION_ATTRIBUTES, *PUSB_NODE_CONNECTION_ATTRIBUTES;
```

## Members

**ConnectionIndex**

On input to the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) I/O control request, this member contains the number of the port.

**ConnectionStatus**

On output from the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) I/O control request, this member contains a [USB\\_CONNECTION\\_STATUS](#) enumerator that indicates the connection status.

**PortAttributes**

On output from the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#) I/O control request, this member contains the Microsoft-extended port attributes.

For Windows Vista, Windows Server 2008, and Windows 7 the Microsoft-extended port attributes field will always be zero.

For Windows XP and Windows Server 2003, **PortAttributes** value might be set to the Microsoft-extended port attributes, [USB\\_PORTATTR\\_NO\\_OVERCURRENT\\_UI](#). This

attribute indicates that no user-visible interface will be displayed when overcurrent occurs on the port.

## Requirements

 Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_ATTRIBUTES](#)

[USB Structures](#)

[USB\\_CONNECTION\\_STATUS](#)

# USB\_NODE\_CONNECTION\_DRIVERKEY\_NAME structure (usbioc.h)

Article05/15/2024

The **USB\_NODE\_CONNECTION\_DRIVERKEY\_NAME** structure is used with the **IOCTL\_USB\_GET\_NODE\_CONNECTION\_DRIVERKEY\_NAME** I/O control request to retrieve the driver key name for the device that is connected to the indicated port.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_DRIVERKEY_NAME {
    ULONG ConnectionIndex;
    ULONG ActualLength;
    WCHAR DriverKeyName[1];
} USB_NODE_CONNECTION_DRIVERKEY_NAME, *PUSB_NODE_CONNECTION_DRIVERKEY_NAME;
```

## Members

**ConnectionIndex**

On input, the port number that the device is connected to.

**ActualLength**

On output, the length of this structure, in bytes.

**DriverKeyName[1]**

On output, the driver key name for the device that is attached to the port that is indicated by **ConnectionIndex**. This name is represented as a Unicode string.

## Requirements

[+] Expand table

Requirement	Value
Header	usbioc.h (include Usbioc.h)

## See also

- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_DRIVERKEY\\_NAME](#)
  - [USB Structures](#)
- 

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USB\_NODE\_CONNECTION\_INFORMATION structure (usbiioctl.h)

Article09/01/2022

The **USB\_NODE\_CONNECTION\_INFORMATION** structure is used with the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#) request to retrieve information about a USB port and connected device.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_INFORMATION {
    ULONG             ConnectionIndex;
    USB_DEVICE_DESCRIPTOR DeviceDescriptor;
    UCHAR             CurrentConfigurationValue;
    BOOLEAN           LowSpeed;
    BOOLEAN           DeviceIsHub;
    USHORT            DeviceAddress;
    ULONG             NumberOfOpenPipes;
    USB_CONNECTION_STATUS ConnectionStatus;
    USB_PIPE_INFO     PipeList[0];
} USB_NODE_CONNECTION_INFORMATION, *PUSB_NODE_CONNECTION_INFORMATION;
```

## Members

`ConnectionIndex`

A value that is greater than or equal to 1 that specifies the number of the port.

`DeviceDescriptor`

A [USB\\_DEVICE\\_DESCRIPTOR](#) structure that reports the USB device descriptor that is returned by the attached device during enumeration.

`CurrentConfigurationValue`

Contains the ID used with the SetConfiguration request to specify that current configuration of the device connected to the indicated port. For an explanation of this value, see section 9.4.7 in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#).

### `LowSpeed`

A Boolean value that indicates whether the port and its connected device are operating at low speed. **TRUE** indicates that the port and its connected device are currently operating at a low speed. **FALSE** indicates otherwise.

### `DeviceIsHub`

A Boolean value that indicates if the device that is attached to the port is a hub. If **TRUE**, the device that is attached to the port is a hub. If **FALSE**, the device is not a hub.

### `DeviceAddress`

The USB-assigned, bus-relative address of the device that is attached to the port.

### `NumberOfOpenPipes`

The number of open USB pipes that are associated with the port.

### `ConnectionStatus`

A [USB\\_CONNECTION\\_STATUS](#)-typed enumerator that indicates the connection status.

### `PipeList[0]`

An array of [USB\\_PIPE\\_INFO](#) structures that describes the open pipes that are associated with the port. Pipe descriptions include the schedule offset of the pipe and the associated endpoint descriptor. This information can be used to calculate bandwidth usage.

## Remarks

If there is no device connected to the USB port,

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#) returns only information about the port. If a device is connected to the port,

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#) returns information about both the port and the connected device.

The [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure is an extended version of [USB\\_NODE\\_CONNECTION\\_INFORMATION](#). The two structures are identical, except for one member. In [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#), the **LowSpeed** member is replaced by the **Speed** member. **LowSpeed** is a Boolean value, so when it is **TRUE**, the device is low speed. When it is **FALSE**, the device is high speed or full speed. Thus the

`USB_NODE_CONNECTION_INFORMATION` structure cannot differentiate between high and full speeds.

The `Speed` member of the `USB_NODE_CONNECTION_INFORMATION_EX` structure is a UCHAR and it can specify any of the values of the `USB_DEVICE_SPEED` enumerator.

## Requirements

Header	usbioclt.h (include Usbioclt.h)
--------	---------------------------------

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION](#)

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

[USB Structures](#)

[USB\\_CONNECTION\\_STATUS](#)

[USB\\_DEVICE\\_DESCRIPTOR](#)

[USB\\_DEVICE\\_SPEED](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

[USB\\_PIPE\\_INFO](#)

# USB\_NODE\_CONNECTION\_INFORMATION\_EX structure (usbioc.h)

Article12/15/2023

The **USB\_NODE\_CONNECTION\_INFORMATION\_EX** structure is used in conjunction with the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) request to obtain information about the connection associated with the indicated USB port.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_INFORMATION_EX {
    ULONG             ConnectionIndex;
    USB_DEVICE_DESCRIPTOR DeviceDescriptor;
    UCHAR             CurrentConfigurationValue;
    UCHAR             Speed;
    BOOLEAN           DeviceIsHub;
    USHORT            DeviceAddress;
    ULONG             NumberOfOpenPipes;
    USB_CONNECTION_STATUS ConnectionStatus;
    USB_PIPE_INFO      PipeList[0];
} USB_NODE_CONNECTION_INFORMATION_EX, *PUSB_NODE_CONNECTION_INFORMATION_EX;
```

## Members

**ConnectionIndex**

Contains a value greater than or equal to 1 that specifies the number of the port.

**DeviceDescriptor**

Contains a structure of type [USB\\_DEVICE\\_DESCRIPTOR](#) that reports the USB device descriptor returned by the attached device during enumeration.

**CurrentConfigurationValue**

Contains the ID used with the SetConfiguration request to specify that current configuration of the device connected to the indicated port. For an explanation of this value, see section 9.4.7 in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#).

### Speed

Contains a value of type [USB\\_DEVICE\\_SPEED](#) that indicates the speed of the device.

### DeviceIsHub

Indicates, when **TRUE**, that the device attached to the port is a hub.

### DeviceAddress

Contains the USB-assigned, bus-relative address of the device that is attached to the port.

### NumberOfOpenPipes

Indicates the number of open USB pipes associated with the port.

### ConnectionStatus

Contains an enumerator of type [USB\\_CONNECTION\\_STATUS](#) that indicates the connection status.

### PipeList[0]

Contains an array of structures of type [USB\\_PIPE\\_INFO](#) that describes the open pipes associated with the port. Pipe descriptions include the schedule offset of the pipe and the associated endpoint descriptor. This information can be used to calculate bandwidth usage.

## Remarks

If there is no device connected,

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) just returns information about the port. If a device is connected to the port

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) returns information about both the port and the connected device.

The [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure is an extended version of [USB\\_NODE\\_CONNECTION\\_INFORMATION](#). The two structures are identical, except for one member. In the extended structure, the **Speed** member indicates the device speed.

The **Speed** member of the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure is a UCHAR and it can specify any of the values of the [USB\\_DEVICE\\_SPEED](#) enumerator. The **Speed** member supports up to *UsbHighSpeed* (USB 2.0). To determine if a device

supports *UsbSuperSpeed* (USB 3.0), use the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) structure.

The following C++ code snippet from the [USBView sample](#) demonstrates how to determine if a device supports UsbSuperSpeed (USB 3.0):

C++

```
// Since the USB_NODE_CONNECTION_INFORMATION_EX is used to display
// the device speed, but the hub driver doesn't support indication
// of superspeed, we overwrite the value if the super speed
// data structures are available and indicate the device is operating
// at SuperSpeed.

if (connectionInfoEx->Speed == UsbHighSpeed
    && connectionInfoExV2 != NULL
    && (connectionInfoExV2->Flags.DeviceIsOperatingAtSuperSpeedOrHigher ||
         connectionInfoExV2-
>Flags.DeviceIsOperatingAtSuperSpeedPlusOrHigher))
{
    connectionInfoEx->Speed = UsbSuperSpeed;
}
```

## Requirements

[ ] [Expand table](#)

Header	usbioclt.h (include Usbioclt.h)
--------	---------------------------------

## See also

- [USB Structures](#)
- [USB\\_CONNECTION\\_STATUS](#)
- [USB\\_PIPE\\_INFO](#)
- [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)
- [USB\\_NODE\\_CONNECTION\\_INFORMATION](#)
- [USB\\_DEVICE\\_SPEED](#)
- [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)
- [USB samples](#)
- [USBView sample](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



## Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2 structure (usbioclt.h)

Article04/01/2021

The **USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2** structure is used with the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) I/O control request to retrieve speed information about a Universal Serial Bus (USB) device that is attached to a particular port.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_INFORMATION_EX_V2 {
    ULONG ConnectionIndex;
    ULONG Length;
    USB_PROTOCOLS SupportedUsbProtocols;
    USB_NODE_CONNECTION_INFORMATION_EX_V2_FLAGS Flags;
} USB_NODE_CONNECTION_INFORMATION_EX_V2,
*PUSB_NODE_CONNECTION_INFORMATION_EX_V2;
```

## Members

**ConnectionIndex**

The port number. If there are *n* ports on the USB hub, the ports are numbered from 1 to *n*. To get the number of ports, send the [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request. The request retrieves the highest port number on the hub.

**Length**

The number of bytes that are required to hold the **USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2** structure. The value must be set by the caller as input to the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) request.

**SupportedUsbProtocols**

The USB signaling protocols that are supported by the port.

In the caller's [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) request, the caller can set **SupportedUsbProtocols** to a bitwise **OR** of one or more flags defined in [USB\\_PROTOCOLS](#).

Upon completion of the request, **SupportedUsbProtocols** contains flags, which indicate the protocols that are actually supported by the port.

#### Flags

A bitmask that indicates the properties and capabilities of the attached device or port. For more information, see [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2\\_FLAGS](#).

## Requirements

<b>Minimum supported client</b>	Windows 8
<b>Minimum supported server</b>	None supported
<b>Header</b>	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2\\_FLAGS](#)

# USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2\_FLAGS union (usbioctl.h)

Article 02/22/2024

The **USB\_NODE\_CONNECTION\_INFORMATION\_EX\_V2\_FLAGS** union is used to indicate the speed at which a USB 3.0 device is currently operating and whether it can operate at higher speed, when attached to a particular port.

Device speed information is obtained in the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) structure by the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) I/O control request.

Or: the speed in which a device attached to a port is currently operating and at what speeds it is capable of operating.

## Syntax

C++

```
typedef union _USB_NODE_CONNECTION_INFORMATION_EX_V2_FLAGS {
    ULONG ul;
    struct {
        ULONG DeviceIsOperatingAtSuperSpeedOrHigher : 1;
        ULONG DeviceIsSuperSpeedCapableOrHigher : 1;
        ULONG DeviceIsOperatingAtSuperSpeedPlusOrHigher : 1;
        ULONG DeviceIsSuperSpeedPlusCapableOrHigher : 1;
        ULONG ReservedMBZ : 28;
    };
} USB_NODE_CONNECTION_INFORMATION_EX_V2_FLAGS,
*PUSB_NODE_CONNECTION_INFORMATION_EX_V2_FLAGS;
```

## Members

ul

A bitmask that indicates the USB speed of the device that is attached to the port.

DeviceIsOperatingAtSuperSpeedOrHigher

If **TRUE**, the attached device is currently operating at SuperSpeed or a higher speed that is defined by the official USB specification.

`DeviceIsSuperSpeedCapableOrHigher`

If **TRUE**, the attached device is a USB 3.0 device and is capable of operating at SuperSpeed or a higher speed that is defined by the official USB specification.

`DeviceIsOperatingAtSuperSpeedPlusOrHigher`

`DeviceIsSuperSpeedPlusCapableOrHigher`

`ReservedMBZ`

Reserved. Do not use.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2\\_FLAGS](#)

# USB\_NODE\_CONNECTION\_NAME structure (usbioctl.h)

Article02/22/2024

The **USB\_NODE\_CONNECTION\_NAME** structure is used with the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_NAME](#) I/O control request to retrieve the symbolic link of the downstream hub that is attached to the port.

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_NAME {
    ULONG ConnectionIndex;
    ULONG ActualLength;
    WCHAR NodeName[1];
} USB_NODE_CONNECTION_NAME, *PUSB_NODE_CONNECTION_NAME;
```

## Members

**ConnectionIndex**

A value that is greater than or equal to 1 that specifies the number of the port to which the hub is attached.

**ActualLength**

The length, in bytes, of the attached hub's symbolic link.

**NodeName[1]**

A Unicode symbolic link for the downstream hub that is attached to the port that is indicated by **ConnectionIndex**. If there is no attached device, the attached device does not have a symbolic link, or if the device is not a hub, **NodeName[0]** will contain a value of **UNICODE\_NULL**.

## Requirements

 Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_NAME](#)

[USB Structures](#)

# USB\_NODE\_CONNECTION\_SUPERSPEEDPLUS\_INFORMATION structure (usbioc.h)

Article 05/22/2024

## Syntax

C++

```
typedef struct _USB_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION {
    ULONG ConnectionIndex;
    ULONG Length;
    USB_DEVICE_CAPABILITY_SUPERSPEEDPLUS_SPEED RxSuperSpeedPlus;
    ULONG RxLaneCount;
    USB_DEVICE_CAPABILITY_SUPERSPEEDPLUS_SPEED TxSuperSpeedPlus;
    ULONG TxLaneCount;
} USB_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION,
*PUSB_NODE_CONNECTION_SUPERSPEEDPLUS_INFORMATION;
```

## Members

ConnectionIndex

The one based port number.

Length

The length of this structure.

RxSuperSpeedPlus

Current operating speed for receiving lanes.

RxLaneCount

The zero-based number of receiving lanes. For actual lane count, add one.

TxSuperSpeedPlus

Current operating speed for transmitting lanes.

TxLaneCount

The zero-based number of transmitting lanes. For actual lane count, add one.

# Requirements

[Expand table](#)

Requirement	Value
Header	usbioctl.h

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

# USB\_NODE\_INFORMATION structure (usbioc.h)

Article 02/22/2024

The **USB\_NODE\_INFORMATION** structure is used with the [IOCTL\\_USB\\_GET\\_NODE\\_INFORMATION](#) I/O control request to retrieve information about a parent device.

## Syntax

C++

```
typedef struct _USB_NODE_INFORMATION {
    USB_HUB_NODE NodeType;
    union {
        USB_HUB_INFORMATION     HubInformation;
        USB_MI_PARENT_INFORMATION MiParentInformation;
    } u;
} USB_NODE_INFORMATION, *PUSB_NODE_INFORMATION;
```

## Members

`NodeType`

A [USB\\_HUB\\_NODE](#) enumerator that indicates whether the parent device is a hub or a non-hub composite device.

`u`

The members of the `u` union are as follows:

`u.HubInformation`

A [USB\\_HUB\\_INFORMATION](#) structure that contains information about a parent hub device.

`u.MiParentInformation`

A [USB\\_MI\\_PARENT\\_INFORMATION](#) structure that contains information about a parent non-hub, composite device.

## Remarks

A parent device can be either a hub or a composite device. The USB stack treats the interfaces of a composite device as though they were children of the composite device. The **USB\_NODE\_INFORMATION** structure can hold information about either kind of parent device (both hubs and composite devices).

## Requirements

[+] Expand table

Requirement	Value
Header	usbioctl.h (include Usbioctl.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_INFORMATION](#)

[USB Structures](#)

[USB\\_HUB\\_INFORMATION](#)

[USB\\_HUB\\_NODE](#)

[USB\\_MI\\_PARENT\\_INFORMATION](#)

# USB\_NOTIFICATION\_TYPE enumeration (usbioc.h)

Article02/22/2024

Defines values for the type of notification that can be requested by a client driver.

## Syntax

C++

```
typedef enum _USB_NOTIFICATION_TYPE {
    EnumerationFailure,
    InsufficientBandwidth,
    InsufficientPower,
    OverCurrent,
    ResetOvercurrent,
    AcquireBusInfo,
    AcquireHubName,
    AcquireControllerName,
    HubOvercurrent,
    HubPowerChange,
    HubNestedTooDeeply,
    ModernDeviceInLegacyHub
} USB_NOTIFICATION_TYPE;
```

## Constants

[ ] Expand table

<code>EnumerationFailure</code> Reserved.
<code>InsufficientBandwidth</code> Reserved.
<code>InsufficientPower</code> Reserved.
<code>OverCurrent</code> Reserved.

<code>ResetOvercurrent</code>	Reserved.
<code>AcquireBusInfo</code>	Indicates the information about the bus. This is retrieved by the <a href="#">IOCTL_INTERNAL_USB_GET_BUS_INFO</a> request. Also see, <a href="#">USB_BUS_NOTIFICATION</a> .
<code>AcquireHubName</code>	Reserved.
<code>AcquireControllerName</code>	Reserved.
<code>HubOvercurrent</code>	Reserved.
<code>HubPowerChange</code>	Reserved.
<code>HubNestedTooDeeply</code>	Reserved.
<code>ModernDeviceInLegacyHub</code>	Reserved.

## Requirements

[\[\] Expand table](#)

Requirement	Value
Header	usbiocrtl.h

# USB\_PIPE\_INFO structure (usbioclt.h)

Article02/22/2024

The **USB\_PIPE\_INFO** structure is used in conjunction with the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) structure and the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#) request to obtain information about a connection and its associated pipes.

## Syntax

C++

```
typedef struct _USB_PIPE_INFO {
    USB_ENDPOINT_DESCRIPTOR EndpointDescriptor;
    ULONG                 ScheduleOffset;
} USB_PIPE_INFO, *PUSB_PIPE_INFO;
```

## Members

**EndpointDescriptor**

Describes the endpoint descriptor. For more information about the endpoint descriptor, see [USB\\_ENDPOINT\\_DESCRIPTOR](#).

**ScheduleOffset**

Indicates the schedule offset assigned to the endpoint for this pipe. See the remarks section for a discussion of the range of values that this member can take.

## Remarks

The USB specification labels isochronous and interrupt transfers as "periodic," because certain periods of transmission time are set aside for these types of transfers. The port driver further divides these periods into "schedule offsets" and distributes the available offsets between those endpoints that are doing periodic transfers. The number of offsets that are available depends on the period. The following table lists the offset values that are available for each period.

[+] Expand table

<b>Period</b>	<b>Available Offsets</b>
1	0
2	0 to 1
4	0 to 3
8	0 to 7
16	0 to 15
32	0 to 31

## Requirements

[+] Expand table

<b>Requirement</b>	<b>Value</b>
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX](#)

[USB Structures](#)

[USB\\_ENDPOINT\\_DESCRIPTOR](#)

# USB\_PORT\_CONNECTOR\_PROPERTIES structure (usbioctl.h)

Article09/01/2022

The **USB\_PORT\_CONNECTOR\_PROPERTIES** structure is used with the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) I/O control request to retrieve information about a port on a particular SuperSpeed hub.

## Syntax

C++

```
typedef struct _USB_PORT_CONNECTOR_PROPERTIES {
    ULONG          ConnectionIndex;
    ULONG          ActualLength;
    USB_PORT_PROPERTIES UsbPortProperties;
    USHORT         CompanionIndex;
    USHORT         CompanionPortNumber;
    WCHAR          CompanionHubSymbolicLinkName[ 1 ];
} USB_PORT_CONNECTOR_PROPERTIES, *PUSB_PORT_CONNECTOR_PROPERTIES;
```

## Members

**ConnectionIndex**

The port number being queried in the request. **ConnectionIndex** is specified by the caller. If there are *n* ports on the SuperSpeed hub, the ports are numbered from 1 to *n*. To get the number of ports, the caller first sends an [IOCTL\\_USB\\_GET\\_HUB\\_INFORMATION\\_EX](#) I/O control request. The request retrieves the highest port number on the hub.

**ActualLength**

The number of bytes required to hold the entire **USB\_PORT\_CONNECTOR\_PROPERTIES** structure including the string that contains the symbolic link name of the companion hub. That string is stored in the **CompanionHubSymbolicLinkName** member. The **ActualLength** value is returned by the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) request and used by the caller to allocate a buffer to hold the received information. For details, see [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#).

**UsbPortProperties**

The port properties. Upon completion of the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) request, **UsbPortProperties** contains a bitwise **OR** of one or more flags indicating the properties and capabilities of the port. The flags are defined in [USB\\_PORT\\_PROPERTIES](#).

#### CompanionIndex

The index of the companion port that is associated with the port being queried (specified by **ConnectionIndex**). If there are  $n$  companion ports, those ports are indexed from 0 to  $n-1$ .

If a port is mapped to more than one companion port, **CompanionIndex** is incremented on multiple queries to enumerate all companion ports.

For SuperSpeed hubs and xHCI controllers, **CompanionIndex** is always 0. For more information, see Remarks.

#### CompanionPortNumber

The port number of the companion port that is given by **CompanionIndex**. If the port being queried shares a USB connector with a port on another hub, **CompanionPortNumber** indicates the port number of the port on the other hub.

**Note** For root hub of an xHCI controller, the shared port might be on the same hub.

#### CompanionHubSymbolicLinkName[1]

The Unicode string that contains the symbolic link of the companion hub that shares the USB connector. If a companion hub exists, **CompanionPortNumber** is nonzero. Otherwise, **CompanionHubSymbolicLinkName [0]** is **NULL**.

## Remarks

A SuperSpeed 3.0 hub contains two independent hub implementations. One is for USB 2.0 devices, and the hub implementation is similar to existing 2.0 hubs. The other hub is only for SuperSpeed devices. Because the USB 2.0 and 3.0 bus signaling are electrically independent, both of those hubs operate simultaneously. Therefore, when a SuperSpeed hub is connected to the host, Windows enumerates the two hubs independently; one hub is associated with a USB 2.0 port, and the other hub with a USB 3.0 port. Each hub

has its downstream and upstream ports. USB physical connectors are shared between ports that are associated with those two hub implementations.

Similarly, an xHCI controller must be able to handle SuperSpeed, high-speed, full-speed, and low-speed devices. The USB 3.0 specification requires an xHCI controller to contain two independent execution units each for USB 3.0 and USB 2.0 bus speeds. The USB 3.0 execution unit handles SuperSpeed traffic on the bus. The USB 2.0 execution unit must handle low, full, and high speed traffic. That requirement can be met in many ways. For instance, in one implementation, the USB 2.0 execution unit can have either a downstream USB 1.1 execution unit or a downstream USB 2.0 hub. The other execution unit handles SuperSpeed traffic on the bus. For instance, in one implementation, the xHCI controller can have a downstream USB 2.0 hub (instead of a USB 2.0 host controller) with a transaction translator to handle full-speed and low-speed traffic. That downstream hub shares connectors with the ports of the SuperSpeed root hub.

In cases where USB connectors are shared, the port that is being queried through the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) I/O control request is specified by **ConnectionIndex**, and the port that shares the connector is called the *companion port*. Upon completion of the request, the **CompanionIndex**, **CompanionPortNumber**, and **CompanionHubSymbolicLinkName** members of [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#) can be used to determine the port routing in those cases.

If more than one companion port is associated with the port that is being queried, the application can get information about all companion ports by sending the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) I/O control request in a loop and incrementing the **CompanionIndex** value in each iteration. When all of the ports have been enumerated and there is no port associated with the index specified in **CompanionIndex**, the request completes successfully, **CompanionPortNumber** is set to 0, and **CompanionHubSymbolicLinkName** is NULL.

To get information about the operating speed of a device attached to a particular port, the application can send the

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) I/O control request.

## Requirements

Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

[IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

[USB\\_HUB\\_INFORMATION\\_EX](#)

[USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

[USB\\_PORT\\_PROPERTIES](#)

# USB\_PORT\_PROPERTIES union (usbioc.h)

Article 02/22/2024

The **USB\_PORT\_PROPERTIES** union is used to report the capabilities of a Universal Serial Bus (USB) port.

The port capabilities are retrieved in the [USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#) structure by the [IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#) I/O control request.

## Syntax

C++

```
typedef union _USB_PORT_PROPERTIES {
    ULONG ul;
    struct {
        ULONG PortIsUserConnectable : 1;
        ULONG PortIsDebugCapable : 1;
        ULONG PortHasMultipleCompanions : 1;
        ULONG PortConnectorIsTypeC : 1;
        ULONG ReservedMBZ : 28;
    };
} USB_PORT_PROPERTIES, *PUSB_PORT_PROPERTIES;
```

## Members

`ul`

A bitmask that indicates the properties and capabilities of the port.

`PortIsUserConnectable`

If **TRUE**, the port is visible to the user and a USB device can be attached to or detached from the port.

`PortIsDebugCapable`

If **TRUE**, the port supports debugging over a USB connection.

`PortHasMultipleCompanions`

`PortConnectorIsTypeC`

## ReservedMBZ

Reserved. Do not use.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_USB\\_GET\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

[USB\\_PORT\\_CONNECTOR\\_PROPERTIES](#)

# USB\_PROTOCOLS union (usbioc.h)

Article 02/22/2024

The **USB\_PROTOCOLS** union is used to report the Universal Serial Bus (USB) signaling protocols that are supported by the port.

The supported protocols are retrieved in the [USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) structure by the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) I/O control request.

In the [IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#) request, the caller specifies a bitwise **OR** of one or more flags defined in **USB\_PROTOCOLS**. Upon successful completion, the request retrieves flags, which indicate the protocols that are actually supported by the port.

## Syntax

C++

```
typedef union _USB_PROTOCOLS {
    ULONG ul;
    struct {
        ULONG Usb110 : 1;
        ULONG Usb200 : 1;
        ULONG Usb300 : 1;
        ULONG ReservedMBZ : 29;
    };
} USB_PROTOCOLS, *PUSB_PROTOCOLS;
```

## Members

ul

A bitmask that indicates the USB signaling protocols that are supported by the port.

Usb110

If **TRUE**, the port supports the protocols that are defined in the USB 1.1 Specification. This indicates that the port supports full-speed and low-speed operations. **Usb110** is always **TRUE** for high-speed ports because those ports support full-speed and low-speed operations through split transactions and transaction translators.

Usb200

If TRUE, the port supports the protocols that are defined USB 2.0 Specification. This indicates that the port supports high-speed operations.

Usb300

If TRUE, the port supports the protocols that are defined USB 3.0 Specification. This indicates that the port supports SuperSpeed operations.

ReservedMBZ

Reserved. Do not use.

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbiocli.h (include Usbiocli.h)

## See also

[IOCTL\\_USB\\_GET\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

[USB\\_NODE\\_CONNECTION\\_INFORMATION\\_EX\\_V2](#)

# USB\_ROOT\_HUB\_NAME structure (usbioc.h)

Article02/22/2024

The **USB\_ROOT\_HUB\_NAME** structure stores the root hub's symbolic device name.

## Syntax

C++

```
typedef struct _USB_ROOT_HUB_NAME {
    ULONG ActualLength;
    WCHAR RootHubName[1];
} USB_ROOT_HUB_NAME, *PUSB_ROOT_HUB_NAME;
```

## Members

ActualLength

Size of the entire data structure in bytes.

RootHubName[1]

Specifies the Unicode string containing the root hub's symbolic device name.

## Requirements

 Expand table

Requirement	Value
Header	usbioc.h (include Usbioc.h)

## See also

[IOCTL\\_INTERNAL\\_USB\\_GET\\_HUB\\_NAME](#)

[USB Structures](#)

# USB\_START\_TRACKING\_FOR\_TIME\_SYNC\_INFORMATION structure (usbioc.h)

Article 02/22/2024

The input and output buffer for the [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

## Syntax

C++

```
typedef struct _USB_START_TRACKING_FOR_TIME_SYNC_INFORMATION {
    HANDLE TimeTrackingHandle;
    BOOLEAN IsStartupDelayTolerable;
} USB_START_TRACKING_FOR_TIME_SYNC_INFORMATION,
*PUSB_START_TRACKING_FOR_TIME_SYNC_INFORMATION;
```

## Members

TimeTrackingHandle

Registration handle for time sync tracking retrieved through the [IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request. On input, this handle must be set to NULL. On output, the USB driver stack sets this member to the assigned handle.

IsStartupDelayTolerable

On input, the caller must specify whether the initial startup latency of up to 2.048 seconds is tolerable. TRUE indicates that the caller can tolerate the initial startup latency FALSE, the registration is delayed until the USB driver stack is able to detect a valid frame or microframe boundary.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioc.h

## See also

[IOCTL\\_USB\\_START\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

# USB\_STOP\_TRACKING\_FOR\_TIME\_SYNC\_INFORMATION structure (usioctl.h)

Article 02/22/2024

The input buffer for the [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

## Syntax

C++

```
typedef struct _USB_STOP_TRACKING_FOR_TIME_SYNC_INFORMATION {
    HANDLE TimeTrackingHandle;
} USB_STOP_TRACKING_FOR_TIME_SYNC_INFORMATION,
*PUSB_STOP_TRACKING_FOR_TIME_SYNC_INFORMATION;
```

## Members

TimeTrackingHandle

The time tracking handle received in the previous [IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#) request.

## Requirements

[Expand table](#)

Requirement	Value
Header	usioctl.h

## See also

[IOCTL\\_USB\\_STOP\\_TRACKING\\_FOR\\_TIME\\_SYNC](#)

# USB\_TOPOLOGY\_ADDRESS structure (usbioc.h)

Article04/17/2024

The **USB\_TOPOLOGY\_ADDRESS** structure is used with the [IOCTL\\_INTERNAL\\_USB\\_GET\\_TOPOLOGY\\_ADDRESS](#) I/O request to retrieve information about a USB device's location in the USB device tree.

## Syntax

C++

```
typedef struct _USB_TOPOLOGY_ADDRESS {
    ULONG PciBusNumber;
    ULONG PciDeviceNumber;
    ULONG PciFunctionNumber;
    ULONG Reserved;
    USHORT RootHubPortNumber;
    USHORT HubPortNumber[5];
    USHORT Reserved2;
} USB_TOPOLOGY_ADDRESS, *PUSB_TOPOLOGY_ADDRESS;
```

## Members

PciBusNumber

Specifies the PCI bus number of the USB host controller to which the USB device is attached.

PciDeviceNumber

Specifies the PCI device number of the USB host controller to which the USB device is attached.

PciFunctionNumber

Specifies the PCI function number of the USB host controller to which the USB device is attached.

Reserved

RootHubPortNumber

Specifies the root hub port number through which the USB device is connected. The USB device can be connected to the root port directly, or it can be connected through 1 or more external USB hubs to the port.

#### HubPortNumber[5]

An array containing the port number on each external hub (between the root hub and the device) through which the USB device is connected. The first element of the array indicates the port on the hub that is connected directly to the root hub. An array containing all zeros indicates that the device is connected directly to the root hub.

#### Reserved2

## Remarks

The reserved members of this structure must be treated as opaque and are reserved for system use.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Available in Windows Vista and later operating systems.
Header	usbioclt.h (include Usbioclt.h)

## See also

[IOCTL\\_INTERNAL\\_USB\\_GET\\_TOPOLOGY\\_ADDRESS](#)

[USB Structures](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USB\_TRANSPORT\_CHARACTERISTICS structure (usbioctl.h)

Article04/01/2021

Stores the transport characteristics at relevant points in time. This structure is used in the [IOCTL\\_USB\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#) request.

## Syntax

C++

```
typedef struct _USB_TRANSPORT_CHARACTERISTICS {
    ULONG    Version;
    ULONG    TransportCharacteristicsFlags;
    ULONG64  CurrentRoundtripLatencyInMilliseconds;
    ULONG64  MaxPotentialBandwidth;
} USB_TRANSPORT_CHARACTERISTICS, *PUSB_TRANSPORT_CHARACTERISTICS;
```

## Members

**Version**

The version is set to USB\_TRANSPORT\_CHARACTERISTICS\_VERSION\_1.

**TransportCharacteristicsFlags**

A bitmask that indicates to the client driver the transport characteristics that are available and are returned in this structure.

If **USB\_TRANSPORT\_CHARACTERISTICS\_LATENCY\_AVAILABLE**

is set, **CurrentRoundtripLatencyInMilliseconds** contains valid information. Otherwise , it must not be used by the client driver.

If **USB\_TRANSPORT\_CHARACTERISTICS\_BANDWIDTH\_AVAILABLE**

is set, **MaxPotentialBandwidth** contains valid information. Otherwise, it must not be used by the client driver.

**CurrentRoundtripLatencyInMilliseconds**

Contains the current round-trip delay in milliseconds from the time a non-isochronous transfer is received by the USB driver stack to the time that the transfer is completed.

For MA-USB, the underlying network could be WiFi, WiGig, Ethernet etc. The delay can vary depending on the underlying network conditions. A client driver should query the latency periodically or whenever it is notified of a change.

#### **MaxPotentialBandwidth**

Contains the total bandwidth of the host controller's shared transport.

For MA-USB, the underlying network transport could be WiFi, WiGig, Ethernet etc. The total available bandwidth can vary depending on several factors such as the negotiation WiFi channel. A client driver should query the total bandwidth periodically or whenever it is notified of a change.

## Requirements

Minimum supported client	Windows 10, version 1709
Minimum supported server	Windows Server 2016
Header	usbioctl.h

## See also

[IOCTL\\_USB\\_GET\\_TRANSPORT\\_CHARACTERISTICS](#)

# USB\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_NOTIFICATION structure (usbioc.h)

Article02/22/2024

Contains registration information filled when the [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request completes.

## Syntax

C++

```
typedef struct _USB_TRANSPORT_CHARACTERISTICS_CHANGE_NOTIFICATION {
    USB_CHANGE_REGISTRATION_HANDLE Handle;
    USB_TRANSPORT_CHARACTERISTICS  UsbTransportCharacteristics;
} USB_TRANSPORT_CHARACTERISTICS_CHANGE_NOTIFICATION,
*PUSB_TRANSPORT_CHARACTERISTICS_CHANGE_NOTIFICATION;
```

## Members

Handle

An opaque handle for this registration.

UsbTransportCharacteristics

A [USB\\_TRANSPORT\\_CHARACTERISTICS](#) structure that is filled by the USB driver stack with the initial values of the transport characteristics.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioc.h

## See also

[IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

# USB\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_REGISTRATION structure (usbioclt.h)

Article02/22/2024

Contains registration information for the [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request.

## Syntax

C++

```
typedef struct _USB_TRANSPORT_CHARACTERISTICS_CHANGE_REGISTRATION {
    ULONG ChangeNotificationInputFlags;
    USB_CHANGE_REGISTRATION_HANDLE Handle;
    USB_TRANSPORT_CHARACTERISTICS UsbTransportCharacteristics;
} USB_TRANSPORT_CHARACTERISTICS_CHANGE_REGISTRATION,
*PUSB_TRANSPORT_CHARACTERISTICS_CHANGE_REGISTRATION;
```

## Members

`ChangeNotificationInputFlags`

A bitmask set by the client driver to register for change notifications that it is interested in. The following bits are valid:

[+] [Expand table](#)

Value	Meaning
USB_REGISTER_FOR_TRANSPORT_LATENCY_CHANGE (0x1)	The client is notified of changes in transport latency.
USB_REGISTER_FOR_TRANSPORT_BANDWIDTH_CHANGE (0x2)	The client is notified of changes in bandwidth.

`Handle`

An opaque handle for this registration.

`UsbTransportCharacteristics`

A [USB\\_TRANSPORT\\_CHARACTERISTICS](#) structure that is filled by the USB driver stack with the initial values of the transport characteristics.

## Remarks

The registration handle received in this request is valid until the caller sends the [IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request to unregister for notifications.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioclt.h

## See also

[IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

# USB\_TRANSPORT\_CHARACTERISTICS\_CHANGE\_UNREGISTRATION structure (usbioc.h)

Article02/22/2024

Contains unregistration information for the [IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request.

## Syntax

C++

```
typedef struct _USB_TRANSPORT_CHARACTERISTICS_CHANGE_UNREGISTRATION {
    USB_CHANGE_REGISTRATION_HANDLE Handle;
} USB_TRANSPORT_CHARACTERISTICS_CHANGE_UNREGISTRATION,
*PUSB_TRANSPORT_CHARACTERISTICS_CHANGE_UNREGISTRATION;
```

## Members

### Handle

An opaque handle for registration that the client driver obtained in the previous [IOCTL\\_USB\\_REGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#) request.

## Requirements

[+] [Expand table](#)

Requirement	Value
Header	usbioc.h

## See also

[IOCTL\\_USB\\_UNREGISTER\\_FOR\\_TRANSPORT\\_CHARACTERISTICS\\_CHANGE](#)

# usbpmapi.h header

Article01/23/2023

This header is the main include header for client drivers of the USB Policy Manager to monitor the activities of USB Type-C connectors and/or get involved into policy decisions of USB Type-C connectors.

Usbpmapi.h includes:

[UsbCTypes.h](#)

Do not include the preceding header directly. Instead, only include Usbpmapi.h.

For more information, see:

- [Write a USB Type-C Policy Manager client driver](#)
- [Universal Serial Bus \(USB\)](#)

usbpmapi.h contains the following programming interfaces:

## Functions

<a href="#">USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS_INIT</a>
Initializes a <b>USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS</b> structure.
<a href="#">UsbPm_AssignConnectorPowerLevel</a>
Attempts a PD contract renegotiation with the specified voltage/current/power value.
<a href="#">USBPM_CLIENT_CONFIG_EXTRA_INFO_INIT</a>
Initializes a <b>USBPM_CLIENT_CONFIG_EXTRA_INFO</b> structure.
<a href="#">USBPM_CLIENT_CONFIG_INIT</a>
Initializes a <b>USBPM_CLIENT_CONFIG</b> structure.
<a href="#">USBPM_CONNECTOR_PROPERTIES_INIT</a>
Initializes a <b>USBPM_CONNECTOR_PROPERTIES</b> structure.
<a href="#">USBPM_CONNECTOR_STATE_INIT</a>

	Initializes a <b>USBPM_CONNECTOR_STATE_INIT</b> structure.
<a href="#">UsbPm_Deregister</a>	Unregisters the client driver with the Policy Manager.
<a href="#">USBPM_HUB_CONNECTOR_HANDLES_INIT</a>	Initializes a <b>USBPM_HUB_CONNECTOR_HANDLES</b> structure.
<a href="#">USBPM_HUB_PROPERTIES_INIT</a>	Initializes a [ <b>USBPM_HUB_PROPERTIES</b> ] structure.
<a href="#">UsbPm_Register</a>	Registers the client driver with the Policy Manager to report hub arrival/removal and connector state changes.
<a href="#">UsbPm_RetrieveConnectorProperties</a>	Retrieves the properties of a connector. The properties are static information that do not change during the lifecycle of a connector.
<a href="#">UsbPm_RetrieveConnectorState</a>	Retrieves the current state of a connector. Unlike connector properties, state information is dynamic, which can change at runtime.
<a href="#">UsbPm_RetrieveHubConnectorHandles</a>	Retrieves connector handles for all connectors of a hub.
<a href="#">UsbPm_RetrieveHubProperties</a>	Retrieves the properties of a hub. Properties are static information that do not change during the lifecycle of a hub.

## Callback functions

<a href="#">EVT_USBPM_EVENT_CALLBACK</a>	Sends notifications about hub arrival/removal and connector state changes.

# Structures

<a href="#">USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS</a>	Describes the parameters for the <code>UsbPm_AssignConnectorPowerLevel</code> .
<a href="#">USBPM_CLIENT_CONFIG</a>	The configuration structure used in the registering the client driver with the Policy Manager
<a href="#">USBPM_CLIENT_CONFIG_EXTRA_INFO</a>	Contains optional information used to configure the client driver's registration.
<a href="#">USBPM_CONNECTOR_PROPERTIES</a>	Describes the properties of a connector.
<a href="#">USBPM_CONNECTOR_STATE</a>	Describes the state of a connector.
<a href="#">USBPM_EVENT_CALLBACK_PARAMS</a>	Contains the details of the events related to changes in policy manager arrival/removal, hub arrival/removal or connector state change.
<a href="#">USBPM_HUB_CONNECTOR_HANDLES</a>	Stores the connector handles for all connectors on a hub.
<a href="#">USBPM_HUB_PROPERTIES</a>	Properties of a connector hub.

# Enumerations

<a href="#">USBPM_ACCESS_TYPE</a>	Defines the access types for calling Policy Manager functions.
<a href="#">USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT</a>	Defines format values used in <code>USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS</code> .

## [USBPM\\_EVENT\\_TYPE](#)

Defines values for types of events.

# EVT\_USBPM\_EVENT\_CALLBACK callback function (usbpmapi.h)

Article02/22/2024

Sends notifications about hub arrival/removal and connector state changes.

## Syntax

C++

```
EVT_USBPM_EVENT_CALLBACK EvtUsbpmEventCallback;

void EvtUsbpmEventCallback(
    [In] PUSBPM_EVENT_CALLBACK_PARAMS Params
)
{...}
```

## Parameters

[In] Params

A pointer to the caller-supplied [USBPM\\_EVENT\\_CALLBACK\\_PARAMS](#) structure that the client driver fills with event-specific data.

## Return value

None

## Remarks

The client driver registers its implementation of this callback function by setting the appropriate member of [USBPM\\_CLIENT\\_CONFIG](#) and then calling [UsbPm\\_Register](#). Policy Manager can invoke the client driver's implementation before [\[UsbPm\\_Register\]](#) returns.

To stop Policy Manager from invoking the callback function, the client driver must call [UsbPm\\_Deregister](#).

Callback function calls are serialized. Only one call is active at a time.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

## See also

[USBPM\\_EVENT\\_CALLBACK\\_PARAMS](#)

# USBPM\_ACCESS\_TYPE enumeration (usbpmapi.h)

Article02/22/2024

Defines the access types that the client driver can specify in [USBPM\\_CLIENT\\_CONFIG](#) when calling [UsbPm\\_Register](#).

## Syntax

C++

```
typedef enum _USBPM_ACCESS_TYPE {
    UsbPmAccessQuery,
    UsbPmAccessAssignConnectorPowerLevel,
    UsbPmAccessAll
} USBPM_ACCESS_TYPE, *PUSBPM_ACCESS_TYPE;
```

## Constants

[+] Expand table

<code>UsbPmAccessQuery</code> Access for calling <a href="#">UsbPm_Retrieve*</a> functions.
<code>UsbPmAccessAssignConnectorPowerLevel</code> Access for calling <a href="#">UsbPm_AssignConnectorPowerLevelAsync</a> .
<code>UsbPmAccessAll</code> Access to all functions.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27

Requirement	Value
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_ASSIGN\_CONNECTOR\_POWER\_LEVEL\_PARAMS structure (usbpmapi.h)

Article 02/22/2024

Describes the parameters for the [UsbPm\\_AssignConnectorPowerLevel](#) function.

Initialize this structure by calling

[USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS\\_INIT](#).

## Syntax

C++

```
typedef struct _USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS {
    USBC_POWER_ROLE PowerRole;
    USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT Format;
    union {
        struct {
            UINT8 MaximumPdPowerIn500mW;
            USBC_UCSI_SET_POWER_LEVEL_C_CURRENT MaximumTypeCCurrent;
        } Ucsi;
        struct {
            USBC_PD_REQUEST_DATA_OBJECT Rdo;
        } Rdo;
    };
} USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS,
*PUSBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS;
```

## Members

**PowerRole**

The USB Type-C power role of the connector, defined in the [USBC\\_POWER\\_ROLE](#) enumeration.

**Format**

The power level format, defined in the [USBPM\\_ASSIGN\\_POWER\\_LEVEL\\_PARAMS\\_FORMAT](#) enumeration.

**Ucsi**

Description for the UCSI inner structure.

### `Ucsi.MaximumPdPowerIn500mW`

Maximum power in 500mW unit for the connector to provide/consume.

### `Ucsi.MaximumTypeCCurrent`

Maximum current for the connector to provide/consume, defined in the [USBC\\_UCSI\\_SET\\_POWER\\_LEVEL\\_C\\_CURRENT](#) enumeration.

### `Rdo`

Description for the RDO inner structure.

### `Rdo.Rdo`

The PD Request Data Object that has sent to the port partner of this connector. See [USBC\\_PD\\_REQUEST\\_DATA\\_OBJECT](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_ASSIGN\_CONNECTOR\_POWER\_LEVEL\_PARAMS\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS](#) structure. The client driver must call this function before calling [UsbPm\\_AssignConnectorPowerLevel](#).

## Syntax

C++

```
void USBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS_INIT(
    [Out] PUSBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS Params,
    [In]  USBC_POWER_ROLE                         PowerRole,
    [In]  USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT   Format
);
```

## Parameters

[Out] Params

A pointer to a [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS](#) structure to initialize.

[In] PowerRole

The USB Type-C power role of the connector to set. The values are defined in the [USBC\\_POWER\\_ROLE](#) enumeration.

[In] Format

The USB Type-C format, defined in [USBPM\\_ASSIGN\\_POWER\\_LEVEL\\_PARAMS\\_FORMAT](#)

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_ASSIGN\_POWER\_LEVEL\_PARAM\_S\_FORMAT enumeration (usbpmapi.h)

Article02/22/2024

Defines format values used in [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS](#).

## Syntax

C++

```
typedef enum _USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT {
    UsbPmAssignPowerLevelParamsFormatInvalid,
    UsbPmAssignPowerLevelParamsFormatUcsi,
    UsbPmAssignPowerLevelParamsFormatRdo
} USBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT,
*PUSBPM_ASSIGN_POWER_LEVEL_PARAMS_FORMAT;
```

## Constants

[ ] [Expand table](#)

<b>UsbPmAssignPowerLevelParamsFormatInvalid</b> Invalid format.
<b>UsbPmAssignPowerLevelParamsFormatUcsi</b> The format is UCSI.
<b>UsbPmAssignPowerLevelParamsFormatRdo</b> The format is in a PD Request Data Object.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27

Requirement	Value
Minimum UMDF version	2.27
Header	usbpmapi.h

# UsbPm\_AssignConnectorPowerLevel function (usbpmapi.h)

Article 04/30/2024

Attempts a PD contract renegotiation with the specified voltage/current/power value.

## Syntax

C++

```
NTSTATUS UsbPm_AssignConnectorPowerLevel(
    [In] USBPM_CLIENT ClientHandle,
    [In] USBPM_CONNECTOR ConnectorHandle,
    [In] PUSBPM_ASSIGN_CONNECTOR_POWER_LEVEL_PARAMS Params
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

[In] ConnectorHandle

The connector handle provided by Policy Manager when it calls the driver's implementation of [EVT\\_USBPM\\_EVENT\\_CALLBACK](#). The handle is set in the `EventData.ConnectorStateChange.ConnectorHandle` member of the `Params` value.

[In] Params

A pointer to a driver-provided [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS](#) structure that contains the voltage/current/power value to negotiate. Initialize the structure by calling [USBPM\\_ASSIGN\\_CONNECTOR\\_POWER\\_LEVEL\\_PARAMS\\_INIT](#).

Specifying 0 as the voltage/current/power value for a connector in power sink role causes the connector to stop charging.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

## Remarks

The negotiated settings are persistent only as long as the port partner is attached, and is discarded when the port partner is detached. The client driver is required to call this function based on the current connector state, without assuming the previous setting.

If this call succeeds, the request has been accepted but PD contract renegotiation might not be complete. The result of PD contract renegotiation can be either success or failure.

As a result of successful PD contract renegotiation, EVT\_USBPM\_EVENT\_CALLBACK is invoked with a connector state change event. If PD contract renegotiation request fails, the callback function is not invoked. For example, the request gets rejected by the partner, or is no longer valid because the connector state has changed.

In Windows 10, version 1809, the driver can only call this function with the power role of UsbCPowerRoleSink and when a partner is attached to the connector.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

## See also

[UsbPm\\_Register](#)

[EVT\\_USBPM\\_EVENT\\_CALLBACK](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

# USBPM\_CLIENT\_CONFIG structure (usbpmapi.h)

Article02/22/2024

The configuration structure used in the registering the client driver with the Policy Manager. This structure is used in the [UsbPm\\_Register](#) call.

## Syntax

C++

```
typedef struct _USBPM_CLIENT_CONFIG {
    ULONG             Version;
    ULONG             AccessDesired;
    PFN_USBPM_EVENT_CALLBACK EventCallback;
    PVOID             Context;
    PUSBPM_CLIENT_CONFIG_EXTRA_INFO ExtraInfo;
} USBPM_CLIENT_CONFIG, *PUSBPM_CLIENT_CONFIG;
```

## Members

**Version**

Version of this structure.

**AccessDesired**

A bitwise-OR of the [USBPM\\_ACCESS\\_TYPE](#) values.

**EventCallback**

A pointer to the client driver's implementation of the [USBPM\\_EVENT\\_CALLBACK](#) callback function.

**Context**

A driver-defined context structure.

**ExtraInfo**

A pointer to a USBPM\_CLIENT\_CONFIG\_EXTRA\_INFO structure that contains additional information, such as the WDM device object associated with the client driver.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

## See also

[UsbPm\\_Register](#)

# USBPM\_CLIENT\_CONFIG\_EXTRA\_INFO structure (usbpmapi.h)

Article02/22/2024

Contains optional information used to configure the client driver's registration.

## Syntax

C++

```
typedef struct _USBPM_CLIENT_CONFIG_EXTRA_INFO {
    ULONG          Size;
    PDEVICE_OBJECT WdmDeviceObject;
} USBPM_CLIENT_CONFIG_EXTRA_INFO, *PUSBPM_CLIENT_CONFIG_EXTRA_INFO;
```

## Members

### Size

Size of this structure.

### WdmDeviceObject

If the client is a kernel mode driver, this is the kernel mode WDM device for the client driver. In user mode, it is not used.

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CLIENT\_CONFIG\_EXTRA\_INFO\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_CLIENT\\_CONFIG\\_EXTRA\\_INFO](#) structure.

## Syntax

C++

```
void USBPM_CLIENT_CONFIG_EXTRA_INFO_INIT(
    [Out] PUSBPM_CLIENT_CONFIG_EXTRA_INFO ExtraInfo,
    [In]  PDEVICE_OBJECT                 WdmDeviceObject
);
```

## Parameters

[Out] ExtraInfo

A pointer to a [USBPM\\_CLIENT\\_CONFIG\\_EXTRA\\_INFO](#) structure to initialize.

[In] WdmDeviceObject

If the client is a kernel mode driver, it should set it to the kernel mode WDM device object after calling [USBPM\\_CLIENT\\_CONFIG\\_INIT](#). In user mode, it is not used.

## Return value

None

## Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27

Requirement	Value
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CLIENT\_CONFIG\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_CLIENT\\_CONFIG](#) structure. The client driver must call this function before calling [UsbPm\\_Register](#).

## Syntax

C++

```
void USBPM_CLIENT_CONFIG_INIT(
    [Out] PUSBPM_CLIENT_CONFIG ClientConfig,
    [In] ULONG AccessDesired,
    [In] PFN_USBPM_EVENT_CALLBACK EventCallback,
    [In] PUSBPM_CLIENT_CONFIG_EXTRA_INFO ExtraInfo
);
```

## Parameters

[Out] ClientConfig

A pointer to a [USBPM\\_CLIENT\\_CONFIG](#) to initialize.

[In] AccessDesired

A bitwise-OR of the flags defined by the [USBPM\\_ACCESS\\_TYPE](#) enumeration that indicates the type of access the client driver requires.

[In] EventCallback

A pointer to the [EVT\\_USBPM\\_EVENT\\_CALLBACK](#) callback function implemented by the client driver.

[In] ExtraInfo

A pointer to a [USBPM\\_CLIENT\\_CONFIG\\_EXTRA\\_INFO](#) structure that contains optional information such as the WDM device object.

## Return value

None

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CONNECTOR\_PROPERTIES structure (usbpmapi.h)

Article02/22/2024

Describes the properties of a connector. This structure is used in the [UsbPm\\_RetrieveConnectorProperties](#) call.

## Syntax

C++

```
typedef struct _USBPM_CONNECTOR_PROPERTIES {
    USBPM_CONNECTOR ConnectorHandle;
    USBPM_HUB        ParentHubHandle;
    ULONG64          ConnectorId;
    ULONG            SupportedTypeCOperatingModes;
    ULONG            SupportedTypeCSourceCurrentAdvertisements;
    BOOLEAN          IsTypeCAudioAccessorySupported;
    BOOLEAN          IsPdSupported;
    ULONG            SupportedPowerRoles;
} USBPM_CONNECTOR_PROPERTIES, *PUSBPM_CONNECTOR_PROPERTIES;
```

## Members

`ConnectorHandle`

A handle of this connector.

`ParentHubHandle`

The handle of the parent hub to which this connector belongs.

`ConnectorId`

A system-assigned identifier.

`SupportedTypeCOperatingModes`

A bitwise OR of the values defined in the [USBC\\_TYPEC\\_OPERATING\\_MODE](#) enumeration.

`SupportedTypeCSourceCurrentAdvertisements`

A bitwise OR of the values defined in the [USBC\\_CURRENT](#) enumeration.

**IsTypeCAudioAccessorySupported**

Indicates whether audio accessories are supported by this connector.

**IsPdSupported**

Indicates whether PD is supported on this connector.

**SupportedPowerRoles**

A bitwise OR of the values defined in the [USBC\\_POWER\\_ROLE](#) enumeration.

## Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CONNECTOR\_PROPERTIES\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_CONNECTOR\\_PROPERTIES](#) structure. The client driver must call this function before calling [UsbPm\\_RetrieveCoConnectorProperties](#).

## Syntax

C++

```
void USBPM_CONNECTOR_PROPERTIES_INIT(
    [Out] PUSBPM_CONNECTOR_PROPERTIES Properties
);
```

## Parameters

[Out] Properties

A pointer to a [USBPM\\_CONNECTOR\\_PROPERTIES](#) to initialize.

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CONNECTOR\_STATE structure (usbpmapi.h)

Article02/22/2024

Describes the state of a connector. This structure is used in the [UsbPm\\_RetrieveConnectorState](#) call. The client driver must initialize this structure by calling [USBPM\\_CONNECTOR\\_STATE\\_INIT](#).

## Syntax

C++

```
typedef struct _USBPM_CONNECTOR_STATE {
    USBPM_CONNECTOR          ConnectorHandle;
    ULONG                     ChangeToken;
    BOOLEAN                  Attached;
    ULONG                     AttachCount;
    USBC_PARTNER              Partner;
    USBC_CURRENT               CurrentAdvertisement;
    USBC_PD_CONN_STATE        PdConnState;
    USBC_PD_REQUEST_DATA_OBJECT Rdo;
    USBC_DATA_ROLE             DataRole;
    USBC_POWER_ROLE            PowerRole;
    UINT8                     SourceCapsCount;
    USBC_PD_POWER_DATA_OBJECT SourceCaps[USBPM_MAX_CAPS_COUNT];
    UINT8                     SinkCapsCount;
    USBC_PD_POWER_DATA_OBJECT SinkCaps[USBPM_MAX_CAPS_COUNT];
    UINT8                     PartnerSourceCapsCount;
    USBC_PD_POWER_DATA_OBJECT PartnerSourceCaps[USBPM_MAX_CAPS_COUNT];
    ULONG                     PdAlternateModesEnteredCount;
} USBPM_CONNECTOR_STATE, *PUSBPM_CONNECTOR_STATE;
```

## Members

**ConnectorHandle**

A handle to this connector.

**ChangeToken**

The change token number of this state. This value is changed for every connector state change.

**Attached**

Indicates whether this connector is attached.

**AttachCount**

Indicates the number of attache events that have occurred on this connector. If client driver sees the transition from one “Attached” state to another “Attached” state but with different “AttachCount”, it indicates two attaches and with two different port partners.

**Partner**

The type of the port partner, defined in the [USBC\\_PARTNER](#) enumeration.

**CurrentAdvertisement**

The amount of Type-C current advertised.

**PdConnState**

Indicates the PD contract state over the connection on this connector.

**Rdo**

The PD Request Data Object that has sent to the port partner of this connector. See [USBC\\_PD\\_REQUEST\\_DATA\\_OBJECT](#).

**DataRole**

Indicates the USB Type-C data role of the connector, defined in the [USBC\\_DATA\\_ROLE](#) enumeration.

**PowerRole**

Indicates the USB Type-C power role of the connector, defined in the [USBC\\_POWER\\_ROLE](#) enumeration.

**SourceCapsCount**

The number of power data objects in the source caps array.

**SourceCaps[USBPM\_MAX\_CAPS\_COUNT]**

The source capabilities of the connector. See [USBC\\_PD\\_POWER\\_DATA\\_OBJECT](#).

**SinkCapsCount**

The number of power data objects in the corresponding array.

`SinkCaps[USBPM_MAX_CAPS_COUNT]`

The sink capability of the connector. See [USBC\\_PD\\_POWER\\_DATA\\_OBJECT](#).

`PartnerSourceCapsCount`

The number of power data objects in the corresponding array.

`PartnerSourceCaps[USBPM_MAX_CAPS_COUNT]`

The source capabilities of the port partner in the corresponding array. See [USBC\\_PD\\_POWER\\_DATA\\_OBJECT](#).

`PdAlternateModesEnteredCount`

The number of alternate modes entered.

## Requirements

  [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_CONNECTOR\_STATE\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_CONNECTOR\\_STATE\\_INIT](#) structure. The client driver must call this function before calling [UsbPm\\_RetrieveConnectorState](#).

## Syntax

C++

```
void USBPM_CONNECTOR_STATE_INIT(
    PUSBPM_CONNECTOR_STATE ConnectorState
);
```

## Parameters

ConnectorState

A pointer a [USBPM\\_CONNECTOR\\_STATE\\_INIT](#) structure to initialize.

## Return value

None

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# UsbPm\_Deregister function (usbpmapi.h)

Article02/22/2024

Unregisters the client driver with the Policy Manager.

## Syntax

C++

```
NTSTATUS UsbPm_Deregister(
    [In] USBPM_CLIENT ClientHandle
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

## Remarks

[UsbPm\_Deregister] does not return until all outstanding calls to the client driver's callback functions are returned. After **UsbPm\_Deregister** returns, Policy Manager no longer invokes callback functions on the same handle.

The driver typically calls **UsbPm\_Register** in the driver's [EVT\\_WDF\\_DEVICE\\_SELF\\_MANAGED\\_IO\\_INIT](#) and unregisters in [EVT\\_WDF\\_DEVICE\\_SELF\\_MANAGED\\_IO\\_CLEANUP](#) by calling [UsbPm\\_Deregister](#).

## Requirements

[ ] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

## See also

[UsbPm\\_Register](#)

# USBPM\_EVENT\_CALLBACK\_PARAMS structure (usbpmapi.h)

Article02/22/2024

Contains the details of the events related to changes in policy manager arrival/removal, hub arrival/removal or connector state change. This structure is used in the [EVT\\_USBPM\\_EVENT\\_CALLBACK](#) callback function.

## Syntax

C++

```
typedef struct _USBPM_EVENT_CALLBACK_PARAMS {
    USBPM_EVENT_TYPE EventType;
    union {
        struct {
            ULONG AccessGranted;
        } PolicyManagerArrival;
        struct {
            USBPM_HUB HubHandle;
        } HubArrivalRemoval;
        struct {
            USBPM_CONNECTOR ConnectorHandle;
        } ConnectorStateChange;
        struct {
            PVOID Context;
        } EventData;
    } USBPM_EVENT_CALLBACK_PARAMS, *PUSBPM_EVENT_CALLBACK_PARAMS;
```

## Members

`EventType`

A [USBPM\\_EVENT\\_TYPE](#)-type value that indicates the type of event.

`EventData`

A union that contains the event-specific data. The client driver should set the event data in the inner structure related to the event.

`EventData.PolicyManagerArrival`

Data about the Policy Manager arrival event.

`EventData.PolicyManagerArrival.AccessGranted`

`EventData.HubArrivalRemoval`

Data about the hub arrival or removal event.

`EventData.HubArrivalRemoval.HubHandle`

The handle to the connector hub.

`EventData.ConnectorStateChange`

Data about the connector state change event.

`EventData.ConnectorStateChange.ConnectorHandle`

The handle to the connector.

`Context`

The context which is provided by the client driver in a previous call to [UsbPm\\_Register](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_EVENT\_TYPE enumeration (usbpmapi.h)

Article02/22/2024

Defines values for types of events. This enumeration is used in the [EVT\\_USBPM\\_EVENT\\_CALLBACK](#) callback function.

## Syntax

C++

```
typedef enum _USBPM_EVENT_TYPE {
    UsbPmEventPolicyManagerArrival,
    UsbPmEventPolicyManagerRemoval,
    UsbPmEventHubArrival,
    UsbPmEventHubRemoval,
    UsbPmEventConnectorStateChange
} USBPM_EVENT_TYPE;
```

## Constants

[\[\] Expand table](#)

`UsbPmEventPolicyManagerArrival`

The Policy Manager has arrived. This is the first callback event after client registration.

`UsbPmEventPolicyManagerRemoval`

The Policy Manager has left.

`UsbPmEventHubArrival`

A new connector hub has arrived.

`UsbPmEventHubRemoval`

A connector hub has been removed.

`UsbPmEventConnectorStateChange`

The connector state has changed.

## Requirements

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

## See also

[EVT\\_USBPM\\_EVENT\\_CALLBACK](#)

[USBPM\\_EVENT\\_CALLBACK\\_PARAMS](#)

# USBPM\_HUB\_CONNECTOR\_HANDLES structure (usbpmapi.h)

Article02/22/2024

Stores the connector handles for all connectors on a hub. This structure is used in the [UsbPm\\_RetrieveHubConnectorHandles](#) function.

## Syntax

C++

```
typedef struct _USBPM_HUB_CONNECTOR_HANDLES {
    USBPM_HUB        HubHandle;
    ULONG           ConnectorCount;
    USBPM_CONNECTOR *ConnectorHandles;
} USBPM_HUB_CONNECTOR_HANDLES, *PUSBPM_HUB_CONNECTOR_HANDLES;
```

## Members

HubHandle

The handle of this hub.

ConnectorCount

The number of connectors on this hub.

ConnectorHandles

A pointer to a connector handle array of all the connectors on this hub. The array is allocated by the Policy Manager is valid during the life time of the hub. The client driver must not change the array or release the array after use.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809

Requirement	Value
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_HUB\_CONNECTOR\_HANDLES\_INIT function (usbpmapi.h)

Article02/22/2024

Initializes a [USBPM\\_HUB\\_CONNECTOR\\_HANDLES](#) structure.

## Syntax

C++

```
void USBPM_HUB_CONNECTOR_HANDLES_INIT(
    [Out] PUSBPM_HUB_CONNECTOR_HANDLES HubConnectorHandles,
    [In]   ULONG                 ConnectorCount,
    [Out]  USBPM_CONNECTOR      *ConnectorHandlesBuffer
);
```

## Parameters

[Out] HubConnectorHandles

A pointer to a [USBPM\\_HUB\\_CONNECTOR\\_HANDLES](#) structure to initialize.

[In] ConnectorCount

The number of connectors on this hub.

[Out] ConnectorHandlesBuffer

A pointer to a connector handle array of all the connectors on this hub.

## Return value

None

## Requirements

  Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

# USBPM\_HUB\_PROPERTIES structure (usbpmapi.h)

Article 02/22/2024

Properties of a connector hub.

## Syntax

C++

```
typedef struct _USBPM_HUB_PROPERTIES {
    USBPM_HUB ParentHubHandle;
    USBPM_HUB HubHandle;
    ULONG     ConnectorCount;
} USBPM_HUB_PROPERTIES, *PUSBPM_HUB_PROPERTIES;
```

## Members

**ParentHubHandle**

The handle of the parent hub of this hub. This value is NULL when this hub has no parent hub.

**HubHandle**

The handle of this hub.

**ConnectorCount**

The number of connectors on this hub.

## Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27

Requirement	Value
Minimum UMDF version	2.27
Header	usbpmapi.h

# USBPM\_HUB\_PROPERTIES\_INIT function (usbpmapi.h)

Article 02/22/2024

Initializes a [USBPM\\_HUB\\_PROPERTIES](#) structure.

## Syntax

C++

```
void USBPM_HUB_PROPERTIES_INIT(
    [Out] PUSBPM_HUB_PROPERTIES Properties
);
```

## Parameters

[Out] Properties

A pointer to a [USBPM\\_HUB\\_PROPERTIES](#) structure to initialize.

## Return value

None

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib

# UsbPm\_Register function (usbpmapi.h)

Article02/22/2024

Registers the client driver with the Policy Manager to report hub arrival/removal and connector state changes.

## Syntax

C++

```
NTSTATUS UsbPm_Register(
    [In] PUSBPM_CLIENT_CONFIG ClientConfig,
    [Out] USBPM_CLIENT *ClientHandle
);
```

## Parameters

[In] ClientConfig

The pointer to a caller-supplied [USBPM\\_CLIENT\\_CONFIG](#) structure. Initialize the structure by calling macro [USBPM\\_CLIENT\\_CONFIG\\_INIT](#).

[Out] ClientHandle

A pointer to a location that receives a handle to the registration operation.

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

## Remarks

The client driver's callback functions might start getting invoked before [UsbPm\\_Register](#) returns. The *ClientHandle* value is set to a valid value before callback functions are invoked.

The driver typically calls [UsbPm\\_Register](#) in the driver's [EVT\\_WDF\\_DEVICE\\_SELF\\_MANAGED\\_IO\\_INIT](#) and unregisters in [EVT\\_WDF\\_DEVICE\\_SELF\\_MANAGED\\_IO\\_CLEANUP](#) by calling [UsbPm\\_Deregister](#).

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

# UsbPm\_RetrieveConnectorProperties function (usbpmapi.h)

Article02/22/2024

Retrieves the properties of a connector. The properties are static, which do not change during the lifecycle of a connector.

## Syntax

C++

```
NTSTATUS UsbPm_RetrieveConnectorProperties(
    [In]    USBPM_CLIENT             ClientHandle,
    [In]    USBPM_CONNECTOR          ConnectorHandle,
    [Out]   PUSBPM_CONNECTOR_PROPERTIES ConnectorProperties
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

[In] ConnectorHandle

The connector handle provided by Policy Manager when it calls the driver's implementation of [EVT\\_USBPM\\_EVENT\\_CALLBACK](#). The handle is set in the `EventData.ConnectorStateChange.ConnectorHandle` member of the *Params* value.

[Out] ConnectorProperties

A pointer to a driver-provided [USBPM\\_CONNECTOR\\_PROPERTIES](#) structure that receives the connector properties. Initialize the structure by calling [USBPM\\_CONNECTOR\\_PROPERTIES\\_INIT](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

## See also

[UsbPm\\_Register](#)

[EVT\\_USBPM\\_EVENT\\_CALLBACK](#)

# UsbPm\_RetrieveConnectorState function (usbpmapi.h)

Article02/22/2024

Retrieves the current state of a connector. Unlike connector properties, state information is dynamic, which can change at runtime.

## Syntax

C++

```
NTSTATUS UsbPm_RetrieveConnectorState(
    [In] USBPM_CLIENT ClientHandle,
    [In] USBPM_CONNECTOR ConnectorHandle,
    [Out] PUSBPM_CONNECTOR_STATE ConnectorState
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

[In] ConnectorHandle

The connector handle provided by Policy Manager when it calls the driver's implementation of [EVT\\_USBPM\\_EVENT\\_CALLBACK](#). The handle is set in the `EventData.ConnectorStateChange.ConnectorHandle` member of the *Params* value.

[Out] ConnectorState

A pointer to a driver-provided [USBPM\\_CONNECTOR\\_STATE](#) structure that receives the connector state. Initialize the structure by calling [USBPM\\_CONNECTOR\\_STATE\\_INIT](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

# Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

## See also

[UsbPm\\_Register](#)

[EVT\\_USBPM\\_EVENT\\_CALLBACK](#)

# UsbPm\_RetrieveHubConnectorHandles function (usbpmapi.h)

Article02/22/2024

Retrieves connector handles for all connectors of a hub.

## Syntax

C++

```
NTSTATUS UsbPm_RetrieveHubConnectorHandles(
    [In] USBPM_CLIENT ClientHandle,
    [In] USBPM_HUB HubHandle,
    [Out] PUSBPM_HUB_CONNECTOR_HANDLES HubConnectorHandles
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

[In] HubHandle

The handle to the hub.

[Out] HubConnectorHandles

A pointer to a [USBPM\\_HUB\\_CONNECTOR\\_HANDLES](#) structures that contains the connector handles. Initialize this structure by calling [USBPM\\_HUB\\_CONNECTOR\\_HANDLES\\_INIT](#). The array is allocated by the Policy Manager is valid during the life time of the hub. The client driver must not change the array or release the array after use.

## Return value

This function returns NTSTATUS.

## Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

# UsbPm\_RetrieveHubProperties function (usbpmapi.h)

Article 10/21/2021

Retrieves the properties of a hub. The properties are static, which do not change during the lifecycle of a hub.

## Syntax

C++

```
NTSTATUS UsbPm_RetrieveHubProperties(
    [In] USBPM_CLIENT ClientHandle,
    [In] USBPM_HUB HubHandle,
    [Out] PUSBPM_HUB_PROPERTIES HubProperties
);
```

## Parameters

[In] ClientHandle

The handle that the client driver received in a previous call to [UsbPm\\_Register](#).

[In] HubHandle

The hub handle provided by Policy Manager when it calls the driver's implementation of [EVT\\_USBPM\\_EVENT\\_CALLBACK](#). The handle is set in the `EventData.HubArrivalRemoval.HubHandle` member of the `Params` value.

[Out] HubProperties

A pointer to a driver-provided [USBPM\\_HUB\\_PROPERTIES](#) structure that receives the hub properties. Initialize the structure by calling [USBPM\\_HUB\\_PROPERTIES\\_INIT](#).

## Return value

Returns STATUS\_SUCCESS if the operation succeeds. Otherwise, returns an appropriate NTSTATUS value.

# Requirements

Minimum supported client	Windows 10, version 1809
Minimum KMDF version	1.27
Minimum UMDF version	2.27
Header	usbpmapi.h
Library	UsbPmApi.lib
IRQL	PASSIVE_LEVEL

## See also

[UsbPm\\_Register](#)

[EVT\\_USBPM\\_EVENT\\_CALLBACK](#)

# usbspec.h header

Article01/23/2023

This header contains declarations for data structures and enumerations used by a USB client driver.

For more information, see:

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

usbspec.h contains the following programming interfaces:

## Structures

### [USB\\_30\\_HUB\\_DESCRIPTOR](#)

The USB\_30\_HUB\_DESCRIPTOR structure contains a SuperSpeed hub descriptor. For information about the structure members, see Universal Serial Bus Revision 3.0 Specification, 10.13.2.1 Hub Descriptor, Table 10-3. SuperSpeed Hub Descriptor.

### [USB\\_COMMON\\_DESCRIPTOR](#)

The USB\_COMMON\_DESCRIPTOR structure contains the head of the first descriptor that matches the search criteria in a call to USBD\_ParseDescriptors.

### [USB\\_CONFIGURATION\\_DESCRIPTOR](#)

The USB\_CONFIGURATION\_DESCRIPTOR structure is used by USB client drivers to hold a USB-defined configuration descriptor.

### [USB\\_DEVICE\\_CAPABILITY\\_FIRMWARE\\_STATUS\\_DESCRIPTOR](#)

USB FW Update as defined in the USB 3.2 ENGINEERING CHANGE NOTICE.

### [USB\\_DEVICE\\_DESCRIPTOR](#)

The USB\_DEVICE\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined device descriptor.

### [USB\\_DEVICE\\_QUALIFIER\\_DESCRIPTOR](#)

The USB\_DEVICE\_QUALIFIER\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined device qualifier descriptor.

## [USB\\_ENDPOINT\\_DESCRIPTOR](#)

The USB\_ENDPOINT\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined endpoint descriptor.

## [USB\\_HUB\\_DESCRIPTOR](#)

The USB\_HUB\_DESCRIPTOR structure contains a hub descriptor.

## [USB\\_INTERFACE\\_DESCRIPTOR](#)

The USB\_INTERFACE\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined interface descriptor.

## [USB\\_STRING\\_DESCRIPTOR](#)

The USB\_STRING\_DESCRIPTOR structure is used by USB client drivers to hold a USB-defined string descriptor.

## [USB\\_SUPERSPEED\\_ENDPOINT\\_COMPANION\\_DESCRIPTOR](#)

The USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR structure is used by USB client drivers to retrieve a USB-defined SuperSpeed Endpoint Companion descriptor. For more information, see section 9.6.7 and Table 9-20 in the official USB 3.0 specification.

# Enumerations

## [USB\\_DEVICE\\_SPEED](#)

The USB\_DEVICE\_SPEED enumeration defines constants for USB device speeds.

# USB\_30\_HUB\_DESCRIPTOR structure (usbspec.h)

Article02/22/2024

The **USB\_30\_HUB\_DESCRIPTOR** structure contains a SuperSpeed hub descriptor. For information about the structure members, see [Universal Serial Bus Revision 3.0 Specification](#), 10.13.2.1 Hub Descriptor, Table 10-3. SuperSpeed Hub Descriptor.

## Syntax

C++

```
typedef struct _USB_30_HUB_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bNumberOfPorts;
    USHORT wHubCharacteristics;
    UCHAR bPowerOnToPowerGood;
    UCHAR bHubControlCurrent;
    UCHAR bHubHdrDecLat;
    USHORT wHubDelay;
    USHORT DeviceRemovable;
} USB_30_HUB_DESCRIPTOR, *PUSB_30_HUB_DESCRIPTOR;
```

## Members

bLength

The length, in bytes, of the descriptor.

bDescriptorType

The descriptor type. For SuperSpeed hub descriptors, the value must be **USB\_30\_HUB\_DESCRIPTOR\_TYPE** (0x2A).

bNumberOfPorts

The number of ports on the hub.

wHubCharacteristics

The hub characteristics.

**bPowerOnToPowerGood**

The time, in 2-millisecond intervals, that it takes the device to turn on completely.

**bHubControlCurrent**

The maximum current requirements, in milliamperes, of the controller component of the hub.

**bHubHdrDecLat**

The hub packet header decode latency.

**wHubDelay**

The average delay, in nanoseconds, that is introduced by the hub.

**DeviceRemovable**

Indicates whether a removable device is attached to each port.

## Requirements

[ ] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbspec.h

## See also

[USB Structures](#)

[USB\\_HUB\\_INFORMATION\\_EX](#)

# USB\_COMMON\_DESCRIPTOR structure (usbspec.h)

Article 02/22/2024

The **USB\_COMMON\_DESCRIPTOR** structure contains the head of the first descriptor that matches the search criteria in a call to [USBD\\_ParseDescriptors](#).

## Syntax

C++

```
typedef struct _USB_COMMON_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
} USB_COMMON_DESCRIPTOR, *PUSB_COMMON_DESCRIPTOR;
```

## Members

bLength

Specifies the entire length of the descriptor, not of this structure.

bDescriptorType

Specifies the descriptor type code, as assigned by USB, for this descriptor.

## Requirements

  Expand table

Requirement	Value
Header	usbspec.h (include Usb100.h)

## See also

[USB Structures](#)

[USBD\\_ParseDescriptors](#)

# USB\_CONFIGURATION\_DESCRIPTOR structure (usbspec.h)

Article04/01/2021

The **USB\_CONFIGURATION\_DESCRIPTOR** structure is used by USB client drivers to hold a USB-defined configuration descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.3.

## Syntax

C++

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

## Members

**bLength**

Specifies the length, in bytes, of this structure.

**bDescriptorType**

Specifies the descriptor type. Must be set to **USB\_CONFIGURATION\_DESCRIPTOR\_TYPE**.

**wTotalLength**

Specifies the total length, in bytes, of all data for the configuration. The length includes all interface, endpoint, class, or vendor-specific descriptors that are returned with the configuration descriptor.

**bNumInterfaces**

Specifies the total number of interfaces supported by this configuration.

#### bConfigurationValue

Contains the value that is used to select a configuration. This value is passed to the USB SetConfiguration request , as described in version 1.1 of the Universal Serial Bus Specification. The port driver does not currently expose a service that allows higher-level drivers to set the configuration.

#### iConfiguration

Specifies the device-defined index of the string descriptor for this configuration.

#### bmAttributes

Specifies a bitmap to describe behavior of this configuration. The bits are described and set in little-endian order.

Bit	Meaning
0 - 4	Reserved.
5	The configuration supports remote wakeup.
6	The configuration is self-powered and does not use power from the bus.
7	The configuration is powered by the bus.

#### MaxPower

Specifies the power requirements of this device in two-milliamperc units. This member is valid only if bit seven is set in **bmAttributes**.

## Remarks

If **wTotalLength** is greater than the buffer size provided in the URB to hold all descriptors retrieved (interface, endpoint, class, and vendor-defined), incomplete data will be returned. In order to retrieve complete descriptors, the request will need to be re-sent with a larger buffer.

If **bmAttributes** bits six and seven are both set, then the device is powered both by the bus and by a source external to the bus.

Other members that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

# Requirements

Header	usbspec.h (include Usb100.h)
--------	------------------------------

## See also

[USB Structures](#)

[USBD\\_CreateConfigurationRequest](#)

[UsbBuildGetDescriptorRequest](#)

# USB\_DEVICE\_CAPABILITY\_FIRMWARE\_STATUS\_DESCRIPTOR structure (usbspec.h)

Article 02/22/2024

For information about the descriptions see USB the 3.2 ENGINEERING CHANGE NOTICE included in the [USB 3.2 Specification](#).

## Syntax

C++

```
typedef struct _USB_DEVICE_CAPABILITY_FIRMWARE_STATUS_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bDevCapabilityType;
    UCHAR bcdDescriptorVersion;
    union {
        ULONG AsUlong;
        struct {
            ULONG GetFirmwareImageHashSupport : 1;
            ULONG DisallowFirmwareUpdateSupport : 1;
            ULONG Reserved : 30;
        };
    } bmAttributes;
} USB_DEVICE_CAPABILITY_FIRMWARE_STATUS_DESCRIPTOR,
*PUSB_DEVICE_CAPABILITY_FIRMWARE_STATUS_DESCRIPTOR;
```

## Members

bLength

bDescriptorType

bDevCapabilityType

bcdDescriptorVersion

bmAttributes

bmAttributes.AsUlong

bmAttributes.GetFirmwareImageHashSupport

`bmAttributes.DisallowFirmwareUpdateSupport`

`bmAttributes.Reserved`

# Requirements

 Expand table

Requirement	Value
Header	usbspec.h

# USB\_DEVICE\_DESCRIPTOR structure (usbspec.h)

Article04/01/2021

The **USB\_DEVICE\_DESCRIPTOR** structure is used by USB client drivers to retrieve a USB-defined device descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.1.

## Syntax

C++

```
typedef struct _USB_DEVICE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    USHORT idVendor;
    USHORT idProduct;
    USHORT bcdDevice;
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

## Members

bLength

Specifies the length, in bytes, of this descriptor.

bDescriptorType

Specifies the descriptor type. Must be set to **USB\_DEVICE\_DESCRIPTOR\_TYPE**.

bcdUSB

Identifies the version of the USB specification that this descriptor structure complies with. This value is a binary-coded decimal number.

**bDeviceClass**

Specifies the class code of the device as assigned by the USB specification group.

**bDeviceSubClass**

Specifies the subclass code of the device as assigned by the USB specification group.

**bDeviceProtocol**

Specifies the protocol code of the device as assigned by the USB specification group.

**bMaxPacketSize0**

Specifies the maximum packet size, in bytes, for endpoint zero of the device. The value must be set to 8, 16, 32, or 64.

**idVendor**

Specifies the vendor identifier for the device as assigned by the USB specification committee.

**idProduct**

Specifies the product identifier. This value is assigned by the manufacturer and is device-specific.

**bcdDevice**

Identifies the version of the device. This value is a binary-coded decimal number.

**iManufacturer**

Specifies a device-defined index of the string descriptor that provides a string containing the name of the manufacturer of this device.

**iProduct**

Specifies a device-defined index of the string descriptor that provides a string that contains a description of the device.

**iSerialNumber**

Specifies a device-defined index of the string descriptor that provides a string that contains a manufacturer-determined serial number for the device.

**bNumConfigurations**

Specifies the total number of possible configurations for the device.

## Remarks

This structure is used to hold a retrieved USB-defined device descriptor. This information can then be used to further configure or retrieve information about the device. Device descriptors are retrieved by submitting a get-descriptor URB.

The **iManufacturer**, **iProduct**, and **iSerialNumber** values, when returned from the host controller driver, contain index values into an array of string descriptors maintained by the device. To retrieve these strings, a string descriptor request can be sent to the device using these index values.

## Requirements

Header	usbspec.h (include Usb100.h)
--------	------------------------------

## See also

[USB Structures](#)

[UsbBuildDescriptorRequest](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

# USB\_DEVICE\_QUALIFIER\_DESCRIPTOR structure (usbspec.h)

Article04/01/2021

The **USB\_DEVICE\_QUALIFIER\_DESCRIPTOR** structure is used by USB client drivers to retrieve a USB-defined device qualifier descriptor.

## Syntax

C++

```
typedef struct _USB_DEVICE_QUALIFIER_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    UCHAR bNumConfigurations;
    UCHAR bReserved;
} USB_DEVICE_QUALIFIER_DESCRIPTOR, *PUSB_DEVICE_QUALIFIER_DESCRIPTOR;
```

## Members

bLength

Specifies the length, in bytes, of this descriptor.

bDescriptorType

Specifies the descriptor type. Must be set to **USB\_DEVICE\_QUALIFIER\_DESCRIPTOR\_TYPE**.

bcdUSB

Identifies the version of the USB specification that this descriptor structure complies with. This value is a binary-coded decimal number.

bDeviceClass

Specifies the class code of the device as assigned by the USB specification group.

### bDeviceSubClass

Specifies the subclass code of the device as assigned by the USB specification group.

### bDeviceProtocol

Specifies the protocol code of the device as assigned by the USB specification group.

### bMaxPacketSize0

Specifies the maximum packet size, in bytes, for endpoint zero of the device. The value must be set to 8, 16, 32, or 64.

### bNumConfigurations

Specifies the total number of possible configurations for the device.

### bReserved

Reserved.

## Remarks

This structure is similar to [USB\\_DEVICE\\_DESCRIPTOR](#), but it contains only those members that can change when the device switches from full-speed operation to high-speed operation or vice versa. If the device is operating at full speed, querying for this descriptor will contain information about how the device would operate at high-speed. If, on the other hand, the device is operating at high-speed, this descriptor will contain information about how the device would operate at full-speed.

## Requirements

Header	usbspec.h (include Usb200.h)
--------	------------------------------

## See also

[USB Structures](#)

[UsbBuildGetDescriptorRequest](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

# USB\_DEVICE\_SPEED enumeration (usbspec.h)

Article 02/22/2024

The **USB\_DEVICE\_SPEED** enumeration defines constants for USB device speeds.

## Syntax

C++

```
typedef enum _USB_DEVICE_SPEED {
    UsbLowSpeed,
    UsbFullSpeed,
    UsbHighSpeed,
    UsbSuperSpeed
} USB_DEVICE_SPEED;
```

## Constants

[ ] Expand table

<b>UsbLowSpeed</b>	Indicates a low-speed USB 1.1-compliant device.
<b>UsbFullSpeed</b>	Indicates a full-speed USB 1.1-compliant device.
<b>UsbHighSpeed</b>	Indicates a high-speed USB 2.0-compliant device.
<b>UsbSuperSpeed</b>	Indicates a SuperSpeed USB 3.0-compliant device.

## Requirements

[ ] Expand table

Requirement	Value
Header	usbspec.h (include Usbspec.h)

## See also

[USB Constants and Enumerations](#)

# USB\_ENDPOINT\_DESCRIPTOR structure (usbspec.h)

Article 04/01/2021

The **USB\_ENDPOINT\_DESCRIPTOR** structure is used by USB client drivers to retrieve a USB-defined endpoint descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.6.

## Syntax

C++

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bEndpointAddress;
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

## Members

**bLength**

Specifies the length, in bytes, of this descriptor.

**bDescriptorType**

Specifies the descriptor type. Must be set to **USB\_ENDPOINT\_DESCRIPTOR\_TYPE**.

**bEndpointAddress**

Specifies the USB-defined endpoint address. The four low-order bits specify the endpoint number. The high-order bit specifies the direction of data flow on this endpoint: 1 for in, 0 for out.

**bmAttributes**

The two low-order bits specify the endpoint type, one of **USB\_ENDPOINT\_TYPE\_CONTROL**, **USB\_ENDPOINT\_TYPE\_ISOCHRONOUS**,

USB\_ENDPOINT\_TYPE\_BULK, or USB\_ENDPOINT\_TYPE\_INTERRUPT.

#### wMaxPacketSize

Specifies the maximum packet size that can be sent from or to this endpoint.

#### bInterval

The **bInterval** value contains the polling interval for interrupt and isochronous endpoints. For other types of endpoint, this value should be ignored. This value reflects the device's configuration in firmware. Drivers cannot change it.

The polling interval, together with the speed of the device and the type of host controller, determine the frequency with which the driver should initiate an interrupt or an isochronous transfer. The value in **bInterval** does not represent a fixed amount of time. It is a relative value, and the actual polling frequency will also depend on whether the device and the USB host controller operate at low, full or high speed.

If either the host controller or the device operates at low speed, the period of time between interrupt transfers (also known as the polling "period") is measured in units of 1 millisecond frames, and the period is related to the value in **bInterval** as indicated the following table:

Value of <b>bInterval</b>	Polling Period (1-millisecond frames)	Interrupt	Isochronous
0 to 15	8	Supported.	
16 to 35	16	Supported.	
36 to 255	32	Supported.	
> 255	Polling intervals > 255 are forbidden by the USB specification.		

For devices and host controllers that can operate at full speed, the period is measured in units of 1 millisecond frames, and the period is related to the value in **bInterval** as indicated the following table:

Value of <b>bInterval</b>	Polling Period (1-millisecond frames)	Interrupt	Isochronous
1	1	Supported.	Supported.
2 to 3	2	Supported.	Supported.
4 to 7	4	Supported.	Supported.

8 to 15	8	Supported.	Supported.
16 to 31	16	Supported.	Not supported.
32 to 255	32	Supported.	Not supported.
> 255	Polling intervals > 255 are forbidden by the USB specification.		

For devices and host controllers that can operate at high speed, the period is measured in units of microframes. There are eight microframes in each 1 millisecond frame. The period is related to the value in **bInterval** by the formula Period = 2 \*\* (**bInterval** - 1), as indicated the following table:

<b>Value of bInterval</b>	<b>Polling Period (microframes)</b>	<b>Interrupt</b>	<b>Isochronous</b>
1	1	Supported.	Supported.
2	2	Supported.	Supported.
3	4	Supported.	Supported.
4	8	Supported.	Supported.
5	16	Supported.	Not supported.
6	32	Supported.	Not supported.
7 to 255	32	Supported.	Not supported.
> 255	Polling intervals > 255 are forbidden by the USB specification.		

The mappings in the preceding tables between periods and polling intervals are valid in Windows 2000 and later operating systems.

## Requirements

Header	usbspec.h (include Usb100.h)
--------	------------------------------

## See also

[USB Structures](#)

[UsbBuildGetDescriptorRequest](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

# USB\_HUB\_DESCRIPTOR structure (usbspec.h)

Article 02/22/2024

The **USB\_HUB\_DESCRIPTOR** structure contains a hub descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 10.15.2.1.

## Syntax

C++

```
typedef struct _USB_HUB_DESCRIPTOR {
    UCHAR bDescriptorLength;
    UCHAR bDescriptorType;
    UCHAR bNumberOfPorts;
    USHORT wHubCharacteristics;
    UCHAR bPowerOnToPowerGood;
    UCHAR bHubControlCurrent;
    UCHAR bRemoveAndPowerMask[64];
} USB_HUB_DESCRIPTOR, *PUSB_HUB_DESCRIPTOR;
```

## Members

**bDescriptorLength**

The length, in bytes, of the descriptor.

**bDescriptorType**

The descriptor type. For hub descriptors, this value should be 0x29.

**bNumberOfPorts**

The number of ports on the hub.

**wHubCharacteristics**

The hub characteristics. For more information about this member, see Universal Serial Bus Specification.

**bPowerOnToPowerGood**

The time, in 2-millisecond intervals, that it takes the device to turn on completely. For more information about this member, see Universal Serial Bus Specification.

#### bHubControlCurrent

The maximum current requirements, in milliamperes, of the controller component of the hub.

#### bRemoveAndPowerMask[64]

Not currently implemented. Do not use this member.

This member implements DeviceRemovable and PortPwrCtrlMask fields of the hub descriptor. For more information about these fields, see Universal Serial Bus Specification.

## Requirements

[] [Expand table](#)

Requirement	Value
Header	usbspec.h (include Usbioctl.h)

## See also

[USB Structures](#)

[USB\\_HUB\\_INFORMATION](#)

# USB\_INTERFACE\_DESCRIPTOR structure (usbspec.h)

Article 02/22/2024

The **USB\_INTERFACE\_DESCRIPTOR** structure is used by USB client drivers to retrieve a USB-defined interface descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.5.

## Syntax

C++

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

## Members

**bLength**

The length, in bytes, of the descriptor.

**bDescriptorType**

The descriptor type. **bDescriptor** must be set to **USB\_INTERFACE\_DESCRIPTOR\_TYPE**.

**bInterfaceNumber**

The index number of the interface.

**bAlternateSetting**

The index number of the alternate setting of the interface.

### bNumEndpoints

The number of endpoints that are used by the interface, excluding the default status endpoint.

### bInterfaceClass

The class code of the device that the USB specification group assigned.

### bInterfaceSubClass

The subclass code of the device that the USB specification group assigned.

### bInterfaceProtocol

The protocol code of the device that the USB specification group assigned.

### iInterface

The index of a string descriptor that describes the interface. For information about this field, see section 9.6.5 in the "Universal Serial Bus Revision 2.0" specification at [USB Technology](#).

## Requirements

[ ] [Expand table](#)

Requirement	Value
Header	usbspec.h (include Usb100.h)

## See also

[USB Structures](#)

[UsbBuildGetDescriptorRequest](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

# USB\_STRING\_DESCRIPTOR structure (usbspec.h)

Article02/22/2024

The **USB\_STRING\_DESCRIPTOR** structure is used by USB client drivers to hold a USB-defined string descriptor. The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.9.

## Syntax

C++

```
typedef struct _USB_STRING_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;
```

## Members

bLength

Specifies the length, in bytes, of the descriptor.

bDescriptorType

Specifies the descriptor type. Must always be **USB\_STRING\_DESCRIPTOR\_TYPE**.

bString[1]

Pointer to a client-allocated buffer that contains, on return from the host controller driver, a Unicode string with the requested string descriptor.

## Remarks

This structure is used to hold a device, configuration, interface, class, vendor, endpoint, or device string descriptor. The string descriptor provides a human-readable description of the component.

Strings returned in `bString` are in Unicode format and the contents of the strings are device-defined.

## Requirements

 Expand table

Requirement	Value
Header	<code>usbspec.h</code> (include <code>Usbioctl.h</code> )

## See also

[USB Structures](#)

[UsbBuildGetDescriptorRequest](#)

[\\_URB\\_CONTROL\\_DESCRIPTOR\\_REQUEST](#)

# USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR structure (usbspec.h)

Article04/01/2021

The **USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR** structure is used by USB client drivers to retrieve a USB-defined SuperSpeed Endpoint Companion descriptor.

The members of this structure are described in the Universal Serial Bus 3.1 Specification available at [USB Document Library](#). See section 9.6.7.

## Syntax

C++

```
typedef struct _USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bMaxBurst;
    union {
        UCHAR AsUchar;
        struct {
            UCHAR MaxStreams : 5;
            UCHAR Reserved1 : 3;
        } Bulk;
        struct {
            UCHAR Mult : 2;
            UCHAR Reserved2 : 5;
            UCHAR SspCompanion : 1;
        } Isochronous;
    } bmAttributes;
    USHORT wBytesPerInterval;
} USB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR,
*PUSB_SUPERSPEED_ENDPOINT_COMPANION_DESCRIPTOR;
```

## Members

**bLength**

Specifies the length, in bytes, of this descriptor.

**bDescriptorType**

Specifies the descriptor type. Must be set to **USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR**.

`bMaxBurst`

Specifies the maximum number of packets that the endpoint can send or receive as a part of a burst.

`bmAttributes`

`bmAttributes.AsUchar`

Specifies the length of the structures.

`bmAttributes.Bulk`

`bmAttributes.Bulk.MaxStreams`

Specifies the maximum number of streams supported by the bulk endpoint.

`bmAttributes.Bulk.Reserved1`

Reserved. Do not use.

`bmAttributes.Isochronous`

`bmAttributes.Isochronous.Mult`

Specifies a zero-based number that determines the maximum number of packets (`bMaxBurst * (Mult + 1)`) that can be sent to the endpoint within a service interval.

`bmAttributes.Isochronous.Reserved2`

Reserved. Do not use.

`bmAttributes.Isochronous.SspCompanion`

`wBytesPerInterval`

Number of bytes per interval.

## Remarks

A client driver that supports streams associated with a bulk endpoint, uses **USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR** to get the maximum number of streams supported by the endpoint. That information is required by the client driver in an open-streams request. In the request, the specified value for **NumberOfStreams** member of the [\\_URB\\_OPEN\\_STATIC\\_STREAMS](#) structure cannot exceed the **MaxStreams** value reported in

**USB\_SUPERSPEED\_ENDPOINT\_COMPANION\_DESCRIPTOR**. For more information about opening streams, see [How to Open and Close Static Streams in a USB Bulk Endpoint](#).

## Requirements

Minimum supported client	Windows 8
Minimum supported server	None supported
Header	usbspec.h (include Usbspec.h)

## See also

[USB Structures](#)