Writing USB Drivers with WDF

June 3, 2010

Abstract

A primary design goal for WDF was to make the driver models easy to extend to support new types of hardware. The first specialized I/O targets in both UMDF and KMDF are hardware-specific and support USB devices. You can use the USB I/O targets to write a fully functional Windows® driver for a USB device that uses the Windows USB device stack.

This paper provides information about writing UMDF and KMDF drivers for USB devices. It assumes that you are familiar with the WDF driver model.

This information applies to the following operating systems:

Windows Server® 2008 R2

Windows 7

Windows Server 2008  
 Windows Vista®  
 Windows Server 2003  
 Windows XP  
 Windows 2000 (KMDF only)

References and resources discussed here are listed at the end of this paper.  
For comprehensive information about writing WDF drivers, see *Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith, available at <http://www.microsoft.com/MSPress/books/10512.aspx>.

The current version of this paper is maintained on the Web at:   
 http://www.microsoft.com/whdc/driver/wdf/USB\_WDF.mspx

***Disclaimer****: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.*

*This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.*

© 2010 Microsoft Corporation. All rights reserved.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| June 2, 2010 | Added USB continuous reader support for UMDF. Updated Resources section. | | | |
| April 9, 2007 | First publication. | | | |

Contents

[Introduction 3](#_Toc263323539)

[About USB Devices 3](#_Toc263323540)

[Device and Configuration Descriptors 4](#_Toc263323541)

[USB Data Transfer Models 5](#_Toc263323542)

[Specialized USB I/O Targets in WDF 6](#_Toc263323543)

[USB Target Device Objects 7](#_Toc263323544)

[USB Interface Objects 8](#_Toc263323545)

[USB Target Pipe Objects 8](#_Toc263323546)

[How to Configure a USB I/O Target 8](#_Toc263323547)

[UMDF Example: Configure a USB I/O Target 9](#_Toc263323548)

[KMDF Example: Configure a USB I/O Target 12](#_Toc263323549)

[How to Send an I/O Request to a USB I/O Target 15](#_Toc263323550)

[UMDF Example: Send a Synchronous Request to a USB I/O Target 16](#_Toc263323551)

[KMDF Example: Send an Asynchronous Request to a USB I/O Target 17](#_Toc263323552)

[USB Continuous Reader 20](#_Toc263323553)

[UMDF Configuration of a USB Continuous Reader 20](#_Toc263323554)

[KMDF Configuration of a USB Continuous Reader 21](#_Toc263323555)

[Resources 21](#_Toc263323556)

# Introduction

A primary design goal for Windows® Driver Foundation (WDF) was to make the driver models easy to extend to support new types of hardware. The first specialized input/output (I/O) targets in both User-Mode Driver Framework (UMDF) and Kernel-Mode Driver Framework (KMDF) are hardware-specific and support Universal Serial Bus (USB) devices. You can use the USB I/O targets to write a fully functional Windows driver for a USB device that uses the Windows USB device stack.

Although using USB devices is generally easy, programming them can be difficult. Drivers must deal with surprise removal, state management, and cleanup because a user can remove a device at any time. Although the USB driver interface is relatively complex, the UMDF and KMDF implementations present the abstraction in an organized way and simplify many of the routine tasks that USB client drivers must perform.

# About USB Devices

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

In USB terminology, the direction of a transfer is based on the host. Thus, IN always refers to transfers to the host from a device and OUT always refers to transfers from the host to a device. USB devices can also support bi-directional transfers of control data. Figure 1 shows a USB host and a device with three endpoints.



Figure 1. USB abstraction

The three endpoints for the USB device in Figure 1 include an OUT endpoint, an IN endpoint, and an endpoint that is used for bi-directional control transfers.

The endpoints on a device are grouped into functional interfaces, and a set of interfaces makes up a device configuration. For example, a one-touch USB backup device might define one group of endpoints as a HID interface that controls the one-touch backup button and another group of endpoints as an interface that provides the mass storage function for the device. The configuration of the device comprises both of these interfaces.

In addition, an individual interface can have multiple settings. Consider a Bluetooth dongle in which one interface is defined for command, control, and lossless data, and a second interface is defined for lossy voice data. The second interface has several alternate settings that provide increasing levels of voice quality and consume increasing bus bandwidth. Only one of the alternate settings is configured at any given time.

The endpoints in the current alternate setting are associated with pipes and therefore can be targets for I/O. All the other endpoints are just endpoints.

## Device and Configuration Descriptors

Each USB device has a device descriptor that provides device-specific and vendor-specific information, such as the version of the USB specification that the device supports, the device class, and the device name, just to name a few. The device descriptor also contains the number of configurations that the device supports. However, WDF and the built-in Windows USB class drivers support only the first configuration on a device.

Each configuration likewise has a descriptor. The configuration descriptor describes the power and wake-up capabilities of the configuration and includes the number of interfaces in the configuration. A configuration and its interfaces follow these rules:

* A configuration contains one or more interfaces, all of which are concurrently active.
* Each interface can have one or more alternate settings.

An alternate setting is a collection of endpoints. Each alternate setting in an interface can have a different number of endpoints or can have the same number of endpoints that consume varying degrees of bus bandwidth.

* An endpoint can be in only one interface within a configuration, but can be used in multiple alternates within that interface.

All of the endpoints in an alternate setting can be in use concurrently.

Figure 2 shows a hypothetical device configuration.



Figure 2. USB configuration

In the figure, Interface 0 has two alternate settings, either one of which can be active at any time. Interface 1 has only one alternate setting.

During configuration, the driver for the USB device selects one or more or interfaces and an alternate setting for each interface. Each interface includes one or more endpoints. By choosing the interfaces and the setting within each interface, the driver indicates its support for particular device functions.

All devices support endpoint 0. Endpoint 0 is a control endpoint that is used to configure the interfaces.

Most USB devices do not provide multiple interfaces or multiple alternate settings. The OSR USB Fx2 device, for example, has one interface with one alternate setting and three endpoints.

## USB Data Transfer Models

USB supports three types of data transfer I/O models—interrupt, bulk, and isochronous—as well as a separate control I/O model. All USB devices support the control mechanism, but support for data transfer mechanisms is optional. Data transfers use unidirectional endpoints, whereas control transfers use bidirectional endpoints.

The data transfer models have the following characteristics:

* Bulk transfers

Unidirectional with no guaranteed latency or bandwidth and guaranteed error-free delivery.

* Interrupt transfers

Unidirectional with guaranteed latency and error retry.

* Isochronous transfers

Guaranteed bandwidth with bounded latency that provides a constant unidirectional data rate with no error retry, which can lead to data loss.

Each endpoint is associated with a data transfer type. Interrupt, bulk, and isochronous endpoints are unidirectional. Control endpoints are bidirectional and are generally used to enumerate a USB device and select an operational configuration. Every endpoint has a unique address.

WDF supports bulk and interrupt endpoints along with the default control endpoint 0.

To communicate with an isochronous endpoint, a KMDF driver can create a URB, use USBD functions to format the URB, use WDF methods to insert the URB into an I/O request object, and send it to the I/O target by using **WdfRequestSend**. See the lsorwr.c file in the Usbsamp sample for an example.

# Specialized USB I/O Targets in WDF

WDF drivers for USB devices should use the specialized USB I/O target support that both UMDF and KMDF provide. WDF defines three types of objects for use with USB I/O targets:

* A USB target device object represents a USB device and provides methods for retrieving information about the device and sending control requests to the device.
* A USB interface object represents an individual interface and supports methods with which a driver can select an alternate setting and retrieve information about the setting.
* A USB target pipe object represents an individual pipe—that is, an endpoint that is configured in the current alternate setting for an interface.

The interfaces and methods in Table 1 support USB.

Table 1. Interfaces and Methods to Support USB

| **USB abstraction** | **UMDF interface** | **KMDF object type and methods** |
| --- | --- | --- |
| USB device | IWDFUsbTargetDevice | WDFUSBDEVICE WdfUsbTargetDeviceXxx |
| Interface | IWDFUsbInterface | WDFUSBINTERFACE WdfUsbInterfaceXxx |
| Pipe | IWDFUsbTargetPipe | WDFUSBPIPE WdfUsbTargetPipeXxx |

A driver creates target device, interface, and target pipe objects during USB configuration, which typically occurs in the driver’s prepare hardware callback. Consequently:

* A UMDF function driver for a USB device must implement the **IPnpCallbackHardware** interface on the device callback object.
* A KMDF driver must register the *EvtDeviceD0Entry*, *EvtDeviceD0Exit*, and *EvtPrepareHardware* callbacks.

A UMDF driver that uses the USB I/O target must specify **WinUSB** as the value for the I/O dispatcher in the INF file. For more information on I/O dispatchers in UMDF, see the WDK, which is listed in the Resources section.

## USB Target Device Objects

A driver calls the framework to create a USB target device object as follows:

* A UMDF driver queries for the **IWDFUsbTargetFactory** interface on the device object and then uses the returned pointer to call the **CreateUsbTargetDevice** method. This method creates the framework USB target device object and returns a pointer to an **IWDFUsbTargetDevice** interface.

The framework-created USB target device object is already configured with the default configuration and the alternate setting 0 for each interface.

* A KMDF driver calls the **WdfUsbTargetDeviceCreate** method, which returns a handle to a WDFUSBDEVICE object. Unlike most of the other KMDF object creation methods, this method does not require an object configuration structure. After the KMDF driver creates the USB target device, it must call **WdfUsbTargetDeviceSelectConfig** to configure the device.

By default, the USB target device object is a child of the device object.

After the driver creates the USB target device object, the driver calls methods on the object to retrieve the configuration and device descriptors. WDF also supports methods that perform the following types of device-specific requests for the USB device I/O target:

* Format and send device I/O control requests to the control pipe.
* Retrieve other information about the device.
* Reset and cycle power on the port. (KMDF only)
* Format and send WDM URBs. (KMDF only)

A KMDF driver can use the **WdfIoTargetStart** and **WdfIoTargetStop** methods to control USB I/O targets as for any other targets. When a KMDF driver specifies a WDFUSBDEVICE handle in a call to one of these methods, the framework starts or stops all of the currently configured pipes on all of the interfaces for the USB device object. For example, if a driver calls **WdfIoTargetStop** on the WDFUSBDEVICE handle, the framework stops sending I/O requests to the pipes that are part of each interface in addition to stopping I/O to the USB target device itself. Consequently, the driver is not required to iterate through all of the pipes on all of the interfaces to stop or start them individually.

## USB Interface Objects

When the driver configures the device, the framework creates a USB interface object for each interface in the configuration. By default, the USB interface object is a child of the device object. A driver calls a method on the USB target device object to get access to the interface objects, as follows:

* A UMDF driver passes an interface number to **IWDFUsbTargetDevice::RetrieveUsbInterface** to receive a pointer to the **IWDFUsbInterface** interface for that particular USB interface object.
* A KMDF driver passes an interface number to **WdfUsbTargetDeviceGetInterface** to get a handle to the WDFUSBINTERFACE object for a particular interface.

After the driver selects an interface, it can select an alternate setting within that interface and then can retrieve information about the pipes in the setting. By default, the framework uses alternate setting 0 within each interface.

## USB Target Pipe Objects

A pipe is an endpoint that is part of the current interface alternate setting. The framework creates a pipe object for each pipe in the setting, and a driver gets access to the pipes as follows:

* A UMDF driver calls **IWDFUsbInterface::RetrieveUsbPipeObject** on the framework USB interface object to get a pointer to an **IWDFUsbTargetPipe** interface on a particular pipe.
* A KMDF driver calls **WdfUsbInterfaceGetConfiguredPipe** to get a handle to the WDFUSBPIPE object for a particular pipe.

For pipes, WDF supports methods that return information about the pipe configuration, manage I/O on the pipe, and control pipe policy, such as limits on packet size.

The lifetime of a pipe object is tied to that of the currently selected interface setting. UMDF relies on reference counting and the WDF object model to delete the objects when they are no longer being used. KMDF explicitly deletes the pipe objects when the driver selects a new alternate setting.

# How to Configure a USB I/O Target

The function driver for a USB device must configure the device before the device can accept any I/O requests other than requests to the control pipe. Depending on the design of the device, configuration can involve one or more of the following:

* Retrieving information about the current configuration, such as the number of interfaces.
* Retrieving the interface objects.
* Selecting an alternate setting within each interface, if the interface supports more than one setting.
* Retrieving the pipes within each interface.

If your device has a single interface with a single alternate setting, you can skip most of these steps and simply retrieve the pipes.

The examples that follow use the OSR Fx2 device, configured as follows:

|  |  |
| --- | --- |
| Number of configurations | 1 |
| Number of interfaces | 1 |
| Number of alternate settings | 1 |
| Number of endpoints | 3 |
| Data transfer types and directions | Interrupt IN Bulk OUT Bulk IN |

## UMDF Example: Configure a USB I/O Target

A UMDF driver creates a framework USB target object and configures the USB target device as part of its **OnPrepareHardware** method of the **IPnpCallbackHardware** interface on the device object. All of the code in this section appears in the Fx2\_Driver sample’s Device.cpp file. However, the code has been edited for its presentation here.

The driver’s first task is to create a target device object in the framework for the USB device, as the following example shows. The target device object in effect connects the driver to the USB subsystem.

Creating a USB target device object in a UMDF driver

HRESULT hr;

IWDFUsbTargetFactory \* pIUsbTargetFactory = NULL;

IWDFUsbTargetDevice \* pIUsbTargetDevice = NULL;

ULONG length;

UCHAR m\_Speed;

hr = m\_FxDevice->QueryInterface (IID\_PPV\_ARGS(&pIUsbTargetFactory));

if (FAILED(hr)) {

. . . //error handling omitted

}

if (SUCCEEDED(hr)) {

hr = pIUsbTargetFactory->CreateUsbTargetDevice(&pIUsbTargetDevice);

length = sizeof(UCHAR);

hr = m\_pIUsbTargetDevice->RetrieveDeviceInformation(DEVICE\_SPEED,

&length,

&m\_Speed

);

}

To create the USB target device object in the framework, the driver must use the **IWDFUsbTargetFactory** interface. It queries for this interface on the device object and uses the returned pointer to call **CreateUsbTargetDevice** to create the framework device object. **CreateUsbTargetDevice** returns a pointer to a **IWDFUsbTargetDevice** interface. The driver can then call **RetrieveDeviceInformation** to get the speed of the device.

Next the driver determines the number of USB interfaces in the device and retrieves an interface pointer. The following example shows this code.

Retrieving a USB interface in a UMDF driver

IWDFUsbInterface \* pIUsbInterface = NULL;

UCHAR NumEndPoints = 0;

UCHAR NumInterfaces = pIUsbTargetDevice->GetNumInterfaces();

WUDF\_TEST\_DRIVER\_ASSERT(1 == NumInterfaces);

hr = pIUsbTargetDevice->RetrieveUsbInterface(0, &pIUsbInterface);

if (FAILED(hr)) {

. . . //error handling omitted }

NumEndPoints = pIUsbInterface->GetNumEndPoints();

if (NumEndPoints != NUM\_OSRUSB\_ENDPOINTS) {

hr = E\_UNEXPECTED;

}

To find the number of interfaces in the device, the driver calls **IWDFUsbTargetDevice::GetNumInterfaces**, which returns the number of interfaces in the default configuration. This driver assumes that the device has one interface and asserts an error otherwise. Interfaces are numbered starting at zero, so when the driver calls **IWDFUsbTargetDevice::RetrieveUsbInterface**, it passes 0 as the first parameter to direct the framework to return an **IWDFUsbInterface** pointer for the first interface. Next the driver gets the number of endpoints in the interface by calling **IWDFUsbInterface::GetNumEndPoints**.

The driver now has the necessary information to configure the pipes. Remember, a pipe is an endpoint that is used in the current alternate interface setting. The following example shows the code to configure the pipes.

Configuring the USB pipes in a UMDF driver

IWDFUsbTargetPipe \* pIUsbPipe = NULL;

IWDFUsbTargetPipe \* pIUsbInputPipe = NULL;

IWDFUsbTargetPipe \* pIUsbOutputPipe = NULL;

IWDFUsbTargetPipe \* pIUsbInterruptPipe = NULL;

for (UCHAR PipeIndex = 0; PipeIndex < NumEndPoints; PipeIndex++) {

hr = pIUsbInterface->RetrieveUsbPipeObject(PipeIndex, &pIUsbPipe);

if (FAILED(hr)) {

. . . //error handling omitted

}

else {

if ( pIUsbPipe->IsInEndPoint() ) {

if ( UsbdPipeTypeInterrupt == pIUsbPipe->GetType() ) {

pIUsbInterruptPipe = pIUsbPipe;

}

else if ( UsbdPipeTypeBulk == pIUsbPipe->GetType() ) {

pIUsbInputPipe = pIUsbPipe;

}

else {

SAFE\_RELEASE(pIUsbPipe);

}

}

else if ( pIUsbPipe->IsOutEndPoint()

&& (UsbdPipeTypeBulk == pIUsbPipe->GetType()) ) {

pIUsbOutputPipe = pIUsbPipe;

}

else {

SAFE\_RELEASE(pIUsbPipe);

}

}

}

if (NULL == pIUsbInputPipe || NULL == pIUsbOutputPipe) {

hr = E\_UNEXPECTED;

}

The endpoint numbers, also called pipe indexes, start at zero. As the example shows, the driver loops through the endpoints, retrieving a pointer to the **IWDFUsbTargetPipe** interface for the associated pipe and then determining the following information for each pipe:

* Whether this is an input pipe or an output pipe.
* Whether the pipe supports interrupt or bulk transfers.

The driver retrieves the **IWDFUsbTargetPipe** interface pointer by calling **IWDFUsbInterface::RetrieveUsbPipeObject**. The driver uses the returned pointer to call the **IWDFUsbTargetPipe::IsInEndPoint**, **IsOutEndPoint**, and **GetType** methods. The driver is designed for the OSR USB Fx2 device, so it expects to find an interrupt IN pipe, a bulk IN pipe, and a bulk OUT pipe.

**IsInEndPoint** returns TRUE for an IN pipe, and the driver calls the **GetType** method to determine the type of data transfer that the pipe supports. **GetType** returns one of the following USBD\_PIPE\_TYPE values:

* **UsbdPipeTypeControl**
* **UsbdPipeTypeIsochronous**
* **UsbdPipeTypeBulk**
* **UsbdPipeTypeInterrupt**

If the pipe supports interrupt or bulk data transfers, the driver saves the pointer to the target pipe interface as pIUsbInterruptPipe or pIUsbInputPipe, respectively.

If this is the bulk OUT pipe, the driver saves the pointer to the target pipe interface as pIUsbOutputPipe.

At the end of the loop, if the driver has not found the expected pipes, it sets an error status.

The driver now configures the pipes, as the following example shows.

Configuring USB pipes in a UMDF driver

LONG timeout;

timeout = ENDPOINT\_TIMEOUT;

hr = m\_pIUsbInputPipe->SetPipePolicy( PIPE\_TRANSFER\_TIMEOUT,

sizeof(timeout),

&timeout

);

if (FAILED(hr)) {

. . . //error handling omitted

}

hr = m\_pIUsbOutputPipe->SetPipePolicy( PIPE\_TRANSFER\_TIMEOUT,

sizeof(timeout),

&timeout

);

if (FAILED(hr)) {

. . . //error handling omitted

}

UMDF supports pipe policy settings to control numerous aspects of device operation, including time-out values and how the device responds to stalled data transfers, among several others. The WinUSB Winusbio.h header file defines the constants that identify the policy types.

For information about pipe policy, see the WDK, which is listed in the Resources section.

The Fx2\_Driver sets time-out values for the input and output pipes to the constant ENDPOINT\_TIMEOUT, defined as 10000 (10 seconds) in the Device.h header file. WinUSB cancels transfers that do not complete within the time-out period.

## KMDF Example: Configure a USB I/O Target

A KMDF driver creates and configures a USB I/O target device object in its *EvtDevicePrepareHardware* callback function. The sample code in this section is based on the Osrusbfx2 sample’s Device.c file.

To create a USB I/O target device object, a KMDF driver calls **WdfUsbTargetDeviceCreate** as the following example shows.

Creating a USB target device object in a KMDF driver

NTSTATUS status;

PDEVICE\_CONTEXT pDeviceContext;

WDF\_USB\_DEVICE\_INFORMATION deviceInfo;

pDeviceContext = GetDeviceContext(Device);

status = WdfUsbTargetDeviceCreate(Device,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&pDeviceContext->UsbDevice

);

The **WdfUsbTargetDeviceCreate** method takes as input parameters a handle to the device object and a pointer to a WDF\_OBJECT\_ATTRIBUTES structure and returns a handle to a WDFUSBDEVICE object.

**Select the Configuration.** If the framework successfully creates the USB target device object, the driver selects the device configuration and retrieves information from the device configuration descriptor by calling **WdfUsbTargetDeviceSelectConfig**. This method configures the device, creates WDF USB interface and pipe objects, and returns information about the specified configuration.

**WdfUsbTargetDeviceSelectConfig** requires a WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS structure as an input and output parameter. On input, the WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS structure selects a configuration. On output, the structure contains information about the selected configuration from the device configuration descriptor.

The device configuration descriptor contains information about many aspects of the device, its configurations, their interfaces, and so forth, and KMDF provides great flexibility in the way that a driver configures the device. Consequently, the framework provides several WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS\_INIT\_XXX functions to initialize this structure. Table 2 lists the variations of initialization functions.

Table 2. Initialization Functions for WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS Structure

| Function version | Description |
| --- | --- |
| DECONFIG | Deconfigures the device, thus indicating that no interfaces are selected. |
| INTERFACES\_DESCRIPTORS | Configures the device by specifying a configuration descriptor and an array of interface descriptors. |
| MULTIPLE\_INTERFACES | Configures the device to use multiple interfaces. |
| SINGLE\_INTERFACE | Configures the device to use a single interface. Drivers for most devices can use this function. |
| URB | Configures the device by specifying a URB. |

KMDF provides additional methods that a driver can call to get specific information about a USB device before the device has been configured. Such methods include—but are not limited to—**WdfUsbTargetDeviceRetrieveConfigDescriptor**, **WdfUsbTargetDeviceGetDeviceDescriptor**, and **WdfUsbTargetDeviceGetInterface**.

The following example shows how the driver selects a configuration and retrieves information about it.

Selecting a USB device configuration in a KMDF driver

WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS configParams;

NTSTATUS status;

PDEVICE\_CONTEXT pDeviceContext;

UCHAR numberConfiguredPipes;

pDeviceContext = GetDeviceContext(Device);

WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS\_INIT\_SINGLE\_INTERFACE(&configParams);

status = WdfUsbTargetDeviceSelectConfig( pDeviceContext->UsbDevice,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&configParams

);

if(!NT\_SUCCESS(status)) return status;

pDeviceContext->UsbInterface =

configParams.Types.SingleInterface.ConfiguredUsbInterface;

numberConfiguredPipes =

configParams.Types.SingleInterface.NumberConfiguredPipes;

The Osrusbfx2 driver selects the first interface in the first configuration descriptor on the USB device by calling **WdfUsbTargetDeviceSelectConfig** and passing a WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS structure.

The driver initializes the structure by using the SINGLE\_INTERFACE variation of the WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS\_INIT function. This function indicates that the device has a single USB interface.

**WdfUsbTargetDeviceSelectConfig** returns a handle to the selected interface in the **Types.SingleInterface.ConfiguredUsbInterface** field of the WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS structure. The driver saves this handle in the UsbInterface field of the device context area.

The OSR USB Fx2 device has only one interface with one alternate setting, so the driver is not required to select an alternate setting.

**Enumerate the Pipes.** Every interface is associated with one or more alternate settings, and each alternate setting is associated with one or more endpoints. Each endpoint in the selected setting is a unidirectional pipe that can perform specific types of data transfers. In the preceding example, the **WdfUsbTargetDeviceSelectConfig** method creates a WDFUSBPIPE object for each pipe in the interface and returns the number of configured pipes in the **Types.SingleInterface.NumberConfiguredPipes** field of the WDF\_USB\_DEVICE\_SELECT\_CONFIG\_PARAMS structure. The driver saves this value in a local variable named numberConfiguredPipes.

Next, the sample driver enumerates all of the USB pipe handles that are associated with the selected interface, as shown in the following example. This step is not strictly necessary, but it shows how to access the WDFUSBPIPE collection that is associated with a USB interface.

Enumerating pipes for a USB interface in a KMDF driver

WDFUSBPIPE pipe;

WDF\_USB\_PIPE\_INFORMATION pipeInfo;

UCHAR index;

for(index=0; index < numberConfiguredPipes; index++) {

WDF\_USB\_PIPE\_INFORMATION\_INIT(&pipeInfo);

pipe = WdfUsbInterfaceGetConfiguredPipe( pDeviceContext->UsbInterface,

index, //PipeIndex,

&pipeInfo

);

// Tell the framework that it's okay to read less than MaximumPacketSize

WdfUsbTargetPipeSetNoMaximumPacketSizeCheck(pipe);

if(WdfUsbPipeTypeInterrupt == pipeInfo.PipeType) {

pDeviceContext->InterruptPipe = pipe;

}

if(WdfUsbPipeTypeBulk == pipeInfo.PipeType

&& WdfUsbTargetPipeIsInEndpoint(pipe)) {

pDeviceContext->BulkReadPipe = pipe;

}

if(WdfUsbPipeTypeBulk == pipeInfo.PipeType

&& WdfUsbTargetPipeIsOutEndpoint(pipe)) {

pDeviceContext->BulkWritePipe = pipe;

}

}

// If we didn't find all 3 pipes, fail the start.

if(!(pDeviceContext->BulkWritePipe

&& pDeviceContext->BulkReadPipe

&& pDeviceContext->InterruptPipe)) {

status = STATUS\_INVALID\_DEVICE\_STATE;

return status;

}

**WdfUsbInterfaceGetConfiguredPipe** requires three parameters:

* A USB interface handle that indicates the interface that contains the pipe.
* The zero-based index of the pipe.
* An optional output parameter that points to storage for a WDF\_USB\_PIPE\_INFORMATION structure.

For each USB interface handle, the framework maintains a collection of configured pipes on the current setting. The index variable value passed as the second parameter is the zero-based index of the pipe about which to get information.

The third parameter is optional and is an output parameter that points to storage for a WDF\_USB\_PIPE\_INFORMATION structure. If the driver supplies the structure, the framework fills it in with information about the specified pipe. The function initializes a WDF\_USB\_PIPE\_INFORMATION structure and passes the address of this structure as the third parameter to **WdfUsbInterfaceGetConfiguredPipe**.

The driver iterates through the collection until it has retrieved information about all the pipes and then ascertains that the pipes match the device configuration that the driver expected to find.

By default, the framework reports an error if a driver uses a read buffer that is not an integral multiple of the pipe’s maximum packet size. This buffer-size check helps to prevent the driver from receiving “babble”—that is, extra data as a result of unexpected bus activity. Within the loop, the driver disables this check by calling **WdfUsbTargetPipeSetNoMaximumPacketSizeCheck** for each pipe.

In addition to the collection of configured pipes, the framework maintains a collection of alternate settings for each interface. Each alternate setting is a collection of endpoints. A driver can get information about this collection and the endpoints in it by calling **WdfUsbInterfaceGetNumEndpoints** and **WdfUsbInterfaceGetEndpointInformation** either before or after configuring the device.

**Get Device Traits.** After it configures the pipes, the Osrusbfx2 driver calls the framework to return additional information about the device, as the following example shows.

Retrieving USB device information in a KMDF driver

WDF\_USB\_DEVICE\_INFORMATION deviceInfo;

ULONG waitWakeEnable;

WDF\_USB\_DEVICE\_INFORMATION\_INIT(&deviceInfo);

status = WdfUsbTargetDeviceRetrieveInformation( pDeviceContext->UsbDevice,

&deviceInfo

);

waitWakeEnable = deviceInfo.Traits & WDF\_USB\_DEVICE\_TRAIT\_REMOTE\_WAKE\_CAPABLE;

if(waitWakeEnable){

status = OsrFxSetPowerPolicy(Device);

if (!NT\_SUCCESS (status)) return status;

}

In this example, the driver initializes a WDF\_USB\_DEVICE\_INFORMATION structure and passes this structure to **WdfUsbTargetDeviceRetrieveInformation** to get information about the device. When the method returns, the structure contains information about the version of USB that the device and its host controller driver (HCD) support, the capabilities of the HCD, and a set of flags that indicate whether the device is self powered, capable of remote wakeup, and operating at high speed.

If the device can support wakeup, the driver enables this feature in its power policy settings. The driver tests the value of the wakeup bit returned in the **Traits** field of the structure and calls a helper function that sets the device power policy if required.

# How to Send an I/O Request to a USB I/O Target

To send an I/O request to a USB I/O target, a driver follows the same steps as for any other I/O target:

1. Create the request or use a request that the framework delivered.

2. Set up the memory objects and buffers for the request.

3. Format the request.

4. Set an I/O completion callback for the request, if appropriate.

5. Send the request.

WDF provides USB-specific methods to format the request, to send certain types of requests, and to retrieve completion parameters.

## UMDF Example: Send a Synchronous Request to a USB I/O Target

To send a device I/O control request to a USB I/O target, a UMDF driver uses the **IWDFIoRequest::Send** method, just as it does to send a request to any other kind of I/O target. The difference is that the driver uses USB-specific UMDF interfaces to format the request.

The **IWDFUsbTargetDevice::FormatRequestForControlTransfer** method for a USB I/O target object formats a request for a USB I/O target. This method takes a pointer to the **IWDFIoRequest** interface for the request, a pointer to a WINUSB\_SETUP\_PACKET structure, a pointer to the **IWDFMemory** interface for the memory object that contains the buffer for the request, and an optional buffer offset.

The WINUSB\_SETUP\_PACKET structure is defined in the Winusbio.h header file that is included in the WDK. The driver fills in the WINUSB\_SETUP\_PACKET structure with information about the request and then calls **FormatRequestForControlTransfer** to format the request as a device I/O control request. If the framework successfully formats the request, the sample driver calls **IWDFIoRequest::Send** to send the request to the USB I/O target.

The following example shows how the Fx2\_Driver sample formats and sends the request.

Sending a device I/O control request to a USB device in a UMDF driver

hr = m\_pIUsbTargetDevice->FormatRequestForControlTransfer(

pWdfRequest,

SetupPacket,

FxMemory,

NULL //TransferOffset

);

}

if (SUCCEEDED(hr)) {

hr = pWdfRequest->Send (m\_pIUsbTargetDevice,

WDF\_REQUEST\_SEND\_OPTION\_SYNCHRONOUS,

0); //Timeout

}

The driver has already created the I/O request object and memory object to use in the request and has saved pointers to their **IWDFIoRequest** and **IWDFMemory** interfaces in pWdfRequest and FxMemory, respectively. The **FormatRequestForControlTransfer** method requires these two pointers along with a pointer to a WINUSB\_SETUP\_PACKET structure. The driver passes NULL for a transfer offset to indicate that the transfer starts at the beginning of the buffer that the memory object describes.

If the framework successfully formats the request, the driver calls **IWDFIoRequest::Send** to send it to the USB I/O target, specifying the flag for a synchronous request.

When the request is complete, the driver retrieves the completion status and USB-specific completion information, as the following example shows.

Retrieving results from a USB I/O request in a UMDF driver

\*LengthTransferred = 0;

IWDFRequestCompletionParams \* FxComplParams = NULL;

IWDFUsbRequestCompletionParams \* FxUsbComplParams = NULL;

pWdfRequest->GetCompletionParams(&FxComplParams);

hr = FxComplParams->GetCompletionStatus();

if (SUCCEEDED(hr)){

HRESULT hrQI =

FxComplParams->QueryInterface(IID\_PPV\_ARGS(&FxUsbComplParams));

FxUsbComplParams->GetDeviceControlTransferParameters( NULL,

LengthTransferred,

NULL,

NULL

);

}

SAFE\_RELEASE(FxUsbComplParams);

SAFE\_RELEASE(FxComplParams);

In the example, the driver calls **IWDFIoRequest::GetCompletionParams** on the request object to get a pointer to the **IWDFRequestCompletionParams** interface for a completion parameters object. With the returned pointer, it can get the completion status for the request by calling **IWDFRequestCompletionParams::GetCompletionStatus**.

If the request completed successfully, the driver queries for the **IWDFUsbRequestCompletionParams** interface that supports methods that provide USB-specific completion data. The **GetDeviceControlTransferParameters** method returns a pointer to the **IWDFMemory** interface for the output buffer, the number of transferred bytes, the offset into the output buffer, and a pointer to the setup packet for the request. The driver is interested only in the number of transferred bytes, so it calls the method with a pointer to a variable to receive that value and NULL for all the other parameters. It then releases the interface pointers for the request and USB parameters.

## KMDF Example: Send an Asynchronous Request to a USB I/O Target

To send a request to a USB I/O target, a KMDF driver uses the methods shown in Table 3.

Table 3. Methods for Sending a Request to a USB I/O Target

| **To send this type of request …** | **Use this method …** |
| --- | --- |
| Cycle power on port (asynchronous) | **WdfUsbTargetDeviceFormatRequestForCyclePort** and **WdfRequestSend** |
| Cycle power on port (synchronous) | **WdfUsbTargetDeviceCyclePortSynchronously** |
| Device I/O control (asynchronous) | **WdfUsbTargetDeviceFormatRequestForControlTransfer** and **WdfRequestSend** |
| Device I/O control (synchronous) | **WdfUsbTargetDeviceSendControlTransferSynchronously** |
| Get string descriptor (synchronous or asynchronous) | **WdfUsbTargetDeviceFormatRequestForString** and **WdfRequestSend** |
| Reset port (synchronous only) | **WdfUsbTargetDeviceResetPortSynchronously** |
| URB (asynchronous) | **WdfUsbTargetDeviceFormatRequestForRead** and **WdfRequestSend** |
| URB (synchronous) | **WdfUsbTargetDeviceSendUrbSynchronously** |

If the driver uses **WdfRequestSend**, it must format the request before sending it by calling a **WdfUsbTargetDeviceFormatXxx** or **WdfUsbTargetPipeFormatXxx** method for the target USB device or pipe, respectively.

Table 4 shows the methods a KMDF driver uses to send a request to a USB target pipe.

Table 4. Methods for Sending a Request to a USB Target Pipe

| **To send this type of request …** | **Use this method …** |
| --- | --- |
| Abort synchronous) | **WdfUsbTargetPipeAbortSynchronously** |
| Abort (asynchronous) | **WdfUsbTargetPipeFormatRequestForAbort** and **WdfRequestSend** |
| Read (asynchronous) | **WdfUsbTargetPipeFormatRequestForRead** and **WdfRequestSend** |
| Read (synchronous) | **WdfUsbTargetPipeReadSynchronously** |
| Reset (asynchronous) | **WdfUsbTargetPipeFormatRequestForReset** and **WdfRequestSend** |
| Reset (synchronous) | **WdfUsbTargetPipeResetSynchronously** |
| URB (asynchronous) | **WdfUsbTargetPipeFormatRequestForUrb** and **WdfRequestSend** |
| URB (synchronous) | **WdfUsbTargetPipeSendUrbSynchronously** |
| Write (asynchronous) | **WdfUsbTargetPipeFormatRequestForWrite** and **WdfRequestSend** |
| Write (synchronous) | **WdfUsbTargetPipeWriteSynchronously** |

The example in this section shows how a driver sends a read request to a USB target pipe and then retrieves the results of the request. The source code is derived from Osrusbfx2\Sys\Final\Bulkrwr.c.

When a read request arrives for the Osrusbfx2 driver, the framework adds it to a sequential queue that the driver created and calls the driver’s *EvtIoRead* callback function. The *EvtIoRead* callback, in turn, sends the request to the USB target pipe. The following example shows the *EvtIoRead* callback function in its entirety, except for some trace statements.

Sending an asynchronous read request to a USB target pipe in a KMDF driver

VOID OsrFxEvtIoRead(

IN WDFQUEUE Queue,

IN WDFREQUEST Request,

IN size\_t Length

)

{

WDFUSBPIPE pipe;

NTSTATUS status;

WDFMEMORY reqMemory;

PDEVICE\_CONTEXT pDeviceContext;

UNREFERENCED\_PARAMETER(Queue);

if (Length > TEST\_BOARD\_TRANSFER\_BUFFER\_SIZE) {

status = STATUS\_INVALID\_PARAMETER;

goto Exit;

}

pDeviceContext = GetDeviceContext(WdfIoQueueGetDevice(Queue));

pipe = pDeviceContext->BulkReadPipe;

status = WdfRequestRetrieveOutputMemory(Request, &reqMemory);

if(!NT\_SUCCESS(status)){

goto Exit;

}

status = WdfUsbTargetPipeFormatRequestForRead(pipe,

Request,

reqMemory,

NULL // Offsets

);

if (!NT\_SUCCESS(status)) {

goto Exit;

}

WdfRequestSetCompletionRoutine( Request,

EvtRequestReadCompletionRoutine,

pipe

);

if (WdfRequestSend(Request,

WdfUsbTargetPipeGetIoTarget(pipe),

WDF\_NO\_SEND\_OPTIONS) == FALSE) {

status = WdfRequestGetStatus(Request);

goto Exit;

}

Exit:

if (!NT\_SUCCESS(status)) {

WdfRequestCompleteWithInformation(Request, status, 0);

}

return;

}

In this example, the driver validates the parameters to ensure that the number of requested bytes does not exceed the capabilities of the device. If the Length parameter is within the correct range, the driver gets a pointer to its device context area, where it has stored a handle to the USB pipe object.

The driver must format the I/O request before sending it to the pipe. Therefore, the driver retrieves the output memory object from the incoming I/O request object and calls **WdfUsbTargetPipeFormatRequestForRead**. The driver passes a handle to the pipe object, a handle to the I/O request object, a handle to the memory object, and an offset in the call.

The driver next sets an I/O completion callback for the request by calling **WdfRequestSetCompletionRoutine**, and then calls **WdfRequestSend** to send the request to the pipe. The driver specifies NO\_SEND\_OPTIONS to indicate that the request should be sent asynchronously and without a time-out. Note that the driver must call **WdfUsbTargetPipeGetIoTarget** to get a handle to the I/O target object for the pipe. The framework creates an I/O target object that is associated with each pipe, but the pipe object itself is not an I/O target.

If **WdfRequestSend** fails, the driver calls **WdfRequestGetStatus** to determine why the operation failed and then completes the I/O request with the failure status.

When the request is complete, the framework calls the I/O completion callback. This callback retrieves the completion parameters and completes the request. The following example shows the I/O completion callback.

Retrieving USB request completion parameters in a KMDF driver

VOID EvtRequestReadCompletionRoutine(

IN WDFREQUEST Request,

IN WDFIOTARGET Target,

PWDF\_REQUEST\_COMPLETION\_PARAMS CompletionParams,

IN WDFCONTEXT Context

)

{

NTSTATUS status;

size\_t bytesRead = 0;

PWDF\_USB\_REQUEST\_COMPLETION\_PARAMS usbCompletionParams;

UNREFERENCED\_PARAMETER(Target);

UNREFERENCED\_PARAMETER(Context);

status = CompletionParams->IoStatus.Status;

usbCompletionParams = CompletionParams->Parameters.Usb.Completion;

bytesRead = usbCompletionParams->Parameters.PipeRead.Length;

WdfRequestCompleteWithInformation(Request, status, bytesRead);

return;

}

The framework calls the I/O completion callback with several parameters, the most interesting of which is a pointer to a WDF\_REQUEST\_COMPLETION\_PARAMS structure. This structure contains a union that supplies the completion parameters for various types of I/O requests. For a request to a USB device or pipe I/O target, the driver uses the **Parameters.Usb** field to access the returned data.

The **Parameters.Usb** field contains a WDF\_USB\_REQUEST\_COMPLETION\_PARAMS structure that is also a union in which each field describes the results for a different combination of request types and target types. To access the results of a read request sent to a pipe, the driver uses the **Parameters.PipeRead** field. Within this field, the **Length** value contains the number of transferred byes. The driver retrieves this value and supplies it in the call to **WdfRequestCompleteWithInformation**.

# USB Continuous Reader

Both UMDF and KMDF provide a continuous reader through which a driver can continuously read data from a USB pipe. The continuous reader ensures that a read request is always available on the pipe and therefore that the driver is always ready to receive data from the device.

## UMDF Configuration of a USB Continuous Reader

A UMDF driver configures a continuous reader for an input pipe by including specialized code in several callbacks:

* The [*IPnpCallbackHardware::OnPrepareHardware*](http://msdn.microsoft.com/en-us/library/dd163499.aspx) callback function must call the [**IWDFUsbTargetPipe2::ConfigureContinuousReader**](http://msdn.microsoft.com/en-us/library/dd435138.aspx) method. This method queues a set of read requests to the device’s I/O target.
* The [*IPnpCallback::OnD0Entry*](http://msdn.microsoft.com/en-us/library/dd183845.aspx) callback function must call [**IWDFIoTargetStateManagement::Start**](http://msdn.microsoft.com/en-us/library/dd183902.aspx) to start the continuous reader.
* The [*IPnpCallback::OnD0Exit*](http://msdn.microsoft.com/en-us/library/dd163502.aspx) callback function must call [**IWDFIoTargetStateManagement::Stop**](http://msdn.microsoft.com/en-us/library/dd183907.aspx) to stop the continuous reader.

Each time that data is available from the device, the I/O target completes a read request and the framework calls one of the following callback functions:

* [*IUsbTargetPipeContinuousReaderCallbackReadComplete::OnReaderCompletion*](http://msdn.microsoft.com/en-us/library/dd435135.aspx), if the I/O target successfully read the data.
* [*IUsbTargetPipeContinuousReaderCallbackReadersFailed::OnReaderFailure*](http://msdn.microsoft.com/en-us/library/dd435133.aspx), if the I/O target reported an error.

## KMDF Configuration of a USB Continuous Reader

A KMDF driver configures a continuous reader for an input pipe by including specialized code in several callbacks:

* The *EvtDevicePrepareHardware* callback function must call the **WdfUsbTargetPipeConfigContinuousReader** method. This method queues a set of read requests to the device’s I/O target.
* The *EvtDeviceD0Entry* callback function must call **WdfIoTargetStart** to start the continuous reader.
* The *EvtDeviceD0Exit* callback function must call **WdfIoTargetStop** to stop the continuous reader.

Each time that data is available from the device, the I/O target completes a read request and the framework calls one of the following callback functions:

* *EvtUsbTargetPipeReadComplete*, if the I/O target successfully read the data.
* *EvtUsbTargetPipeReadersFailed*, if the I/O target reported an error.

# Resources

Windows Driver Foundation on the WHDC website  
<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

Windows Driver Kit  
<http://www.microsoft.com/whdc/DevTools/wdk/default.mspx>

#### White Papers and Books

Architecture of the Kernel-Mode Driver Framework  
[http://www.microsoft.com/whdc/driver/wdf/kmdf-arch.mspx](http://go.microsoft.com/fwlink/?LinkId=55233)

Architecture of the User-Mode Driver Framework  
<http://www.microsoft.com/whdc/driver/wdf/umdf-arch.mspx>

*Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith  
 <http://www.microsoft.com/MSPress/books/10512.aspx>

#### WDK Documentation Topics

ACCESS\_MASK  
<http://msdn.microsoft.com/en-gb/library/ff540466.aspx>

Handling I/O Requests in Framework-based Drivers  
<http://msdn.microsoft.com/en-us/library/ff543296.aspx>

Object Names  
<http://msdn.microsoft.com/en-us/library/ff557762.aspx>

Specifying WDF Directives in INF Files  
<http://msdn.microsoft.com/en-us/library/ff560526.aspx>

WinUsb\_SetPipePolicy Function<http://msdn.microsoft.com/en-gb/library/ff540304.aspx>

WINUSB\_SETUP\_PACKET Structure  
<http://msdn.microsoft.com/en-us/library/ff540313.aspx>

#### WDK Samples

Echo\_driver %wdk%\Src\Umdf\Usb\Echo\_driver

Filter %wdk%\Src\Umdf\Usb\Filter

Fx2\_Driver %wdk%\Src\Umdf\Usb\Fx2\_driver

Kbfiltr %wdk%\Src\Kmdf\Kbfiltr\Sys

Ndisedge %wdk%\Src\Kmdf\Ndisedge\60

Osrusbfx2 %wdk%\Src\Kmdf\Osrusbfx2\Sys\Final

Toastmon %wdk%\Src\Kmdf\Toaster\Toastmon

Usbsamp %wdk%\Src\Kmdf\Usbsamp