Selective Suspend in USB Drivers

November 20, 2009

Abstract

A USB function driver supports runtime idle detection by implementing USB selective suspend. This white paper provides guidance for driver developers on how to implement selective suspend in USB drivers that are based on the Windows® Driver Foundation (WDF).

This information applies to the following operating systems:
 Windows 7
 Windows Server® 2008 R2
 Windows Server 2008
 Windows Vista®
 Windows Server 2003
 Windows XP
 Windows 2000

References and resources discussed here are listed at the end of this paper.

For the latest information, see:
 <http://www.microsoft.com/whdc/driver/wdf/USB_select-susp.mspx>

Disclaimer:

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document History

| **Date** | **Change** |
| --- | --- |
| November 20, 2009 | Describe how a UMDF USB driver can be power policy owner and implement idle and wake support. Update sample locations in WDK. |
| March 20, 2008 | First publication |

Contents

[Introduction 4](#_Toc246322687)

[About Selective Suspend 4](#_Toc246322688)

[Power Policy Ownership, I/O Queues, and Selective Suspend 5](#_Toc246322689)

[Selective Suspend in a UMDF Function Driver 6](#_Toc246322690)

[Guidelines for Selective Suspend in UMDF Drivers 6](#_Toc246322691)

[Power Policy Ownership in UMDF USB Drivers 6](#_Toc246322692)

[I/O Queues in UMDF USB Drivers 7](#_Toc246322693)

[Supporting USB Selective Suspend in a UMDF PPO 8](#_Toc246322694)

[Supporting USB Selective Suspend in a non-PPO UMDF Driver 9](#_Toc246322695)

[System Wake in a UMDF Driver 11](#_Toc246322696)

[Selective Suspend in a KMDF Function Driver 12](#_Toc246322697)

[Guidelines for Selective Suspend in KMDF Drivers 12](#_Toc246322698)

[Power Policy Ownership and KMDF USB Drivers 12](#_Toc246322699)

[I/O Queue Configuration in KMDF Drivers 12](#_Toc246322700)

[Selective Suspend Mechanism for KMDF Function Drivers 12](#_Toc246322701)

[Supporting USB Selective Suspend in a KMDF Function Driver 13](#_Toc246322702)

[Initializing Power Policy Settings in a KMDF Function Driver 13](#_Toc246322703)

[Preventing USB Device Suspension 15](#_Toc246322704)

[Including a Registry Key (HID Drivers Only) 16](#_Toc246322705)

[Remote Wake Support for KMDF Drivers 16](#_Toc246322706)

[Checking Device Capabilities 16](#_Toc246322707)

[Enabling Remote Wakeup 17](#_Toc246322708)

[Resources 18](#_Toc246322709)

# Introduction

Selective suspend is the ability to power down and later resume an idle USB device while the computer to which it is attached remains in the working state (S0). For energy-efficient operation—especially on mobile PCs—all USB devices and drivers should support selective suspend. Powering down a device when it is idle, but while the system remains in the S0 state, has the following significant advantages:

* Selective suspend saves power.
* Selective suspend can help reduce environmental factors such as thermal load and noise.

If your device hardware can power down while it is idle, the driver should support this feature. Selective suspend support in a USB driver that is based on the Windows® Driver Foundation (WDF) requires at most a few extra callbacks beyond those required for basic Plug and Play support.

Every function driver for a USB device should implement aggressive power management that suspends an idle device while the system is running. This paper describes how to implement selective suspend in a WDF-based driver. If you are not familiar with WDF, see the Windows Driver Kit (WDK) and *Developing Drivers with the Windows Driver Foundation*, which are listed in “Resources.”

# About Selective Suspend

USB devices support runtime idle detection through USB selective suspend. Selective suspend allows an idle device to be put into a suspended state without affecting other devices that are connected to the same hub or—in the case of a multifunction device—without affecting the other functions in the device. When all devices or functions have been suspended, the entire hub or multifunction device can be powered down.

From the hardware perspective, selective suspend is a physical state on a USB port. When all functions that are attached to the port are idle, the port can enter selective suspend.

To conform to the USB specification, all USB devices must support selective suspend. When the USB bus is idle, the device must be able to power down. The Microsoft-supplied USB hub drivers implement selective suspend at the hardware level.

USB function drivers should implement selective suspend for their individual device functions through WDF, which communicates with the bus drivers and manages the device I/O control requests that suspend and resume device functions. WDF enables both kernel-mode and user-mode drivers to support selective suspend.

The details of a function driver’s USB selective suspend code depend on whether the driver runs in user mode or kernel mode. Consider these guidelines:

* Use the user-mode driver framework (UMDF) to implement USB drivers whenever possible. User-mode drivers are less likely to corrupt system data and are simpler to debug than kernel-mode drivers.
* Use the kernel-mode driver framework (KMDF) only if the driver streams data through isochronous endpoints or requires other features or resources that are available only in kernel mode.

# Power Policy Ownership, I/O Queues, and Selective Suspend

The power policy owner (PPO) for a device stack is the driver that determines which power state the device should be in at any given time. Only one driver in each device stack can be the PPO. The function driver typically is the PPO for its device.

If your USB driver supports selective suspend and is layered above the PPO in its device stack, the driver must not use power-managed queues. This is true for both UMDF and KMDF drivers. If requests arrive for power-managed queues while the device is suspended, the entire device stack can stall.

Figure 1 shows the flow of I/O requests to a USB driver through its I/O queues.

Figure 1. Flow of Requests to a WDF USB Driver

In the figure, a request arrives for a USB driver. The framework adds the request to the appropriate queue.

If the queue is not power managed, the framework presents the request to the driver according to the dispatch type that the driver configured for the queue (sequential, parallel, or manual). The driver then handles the request.

If the queue is power managed and the device is not suspended, the framework presents the request to the driver according to the configured dispatch type.

However, if the device is suspended, the framework’s actions depend on whether the driver is the PPO for the device stack. If the driver is the PPO, the framework communicates with the USB parent drivers to power up the device. After the device has resumed, the framework presents the request to the driver.

If the driver is not the PPO, the framework takes no further actions because only the PPO can resume the device. The request remains in the queue. The device stack stalls if the PPO does not receive any requests that cause it to resume the device.

# Selective Suspend in a UMDF Function Driver

UMDF function drivers can support USB selective suspend in either of two ways:

* By claiming power policy ownership and handling device idle power-down and resume.
* By relying on the WinUSB.sys driver, which Microsoft supplies, to handle selective suspend. WinUSB.sys is installed as part of the kernel-mode device stack during the installation of the UMDF USB driver. WinUSB.sys implements the underlying mechanisms for suspending and resuming USB device operation.

Both approaches require only small amounts of code. The IdleWake sample that is provided in the WDK shows how to support selective suspend in a UMDF USB driver. You can find this sample in %WinDDK%\*BuildNumber*\Src\Usb\OsrUsbFx2\
UMDF\Fx2\_Driver\IdleWake. The folder contains both PPO and non-PPO versions of the sample.

For more information about UMDF and WinUSB.sys, see “Writing USB Drivers with WDF” on the WHDC Web site.

## Guidelines for Selective Suspend in UMDF Drivers

UMDF drivers that support selective suspend must follow these guidelines:

* The UMDF driver can claim power policy ownership for its device stack, but is not required to do so. By default, the underlying WinUSB.sys driver owns power policy.
* A UMDF driver that supports selective suspend and is the PPO can use power-managed queues or queues that are not power-managed. A UMDF driver that supports selective suspend but is not the PPO must not use power-managed queues.

### Power Policy Ownership in UMDF USB Drivers

By default, WinUSB.sys is the PPO for a device stack that contains a UMDF USB driver. Starting with WDF 1.9, UMDF-based USB drivers can claim power policy ownership. Because only one driver in each device stack can be the PPO, a UMDF USB driver that is the PPO must explicitly disable power policy ownership in WinUSB.sys.

To claim power policy ownership in a UMDF USB driver

1. Call **IWDFDeviceInitialize::SetPowerPolicyOwnership** and pass **TRUE**, typically from the **IDriverEntry::OnDeviceAdd** method on the driver callback object. For example:

FxDeviceInit->SetPowerPolicyOwnership(TRUE);

2. Disable power policy ownership in WinUSB. In the driver’s INF file, include an **AddReg** directive that sets the **WinUsbPowerPolicyOwnershipDisabled** value in the registry to a nonzero value. The **AddReg** directive must appear in a DDInstall.HW section. For example:

[MyDriver\_Install.NT.hw]
AddReg=MyDriver\_AddReg

[MyDriver\_AddReg]
HKR,,"WinUsbPowerPolicyOwnershipDisabled",0x00010001,1

UMDF USB drivers that support selective suspend and are built with WDF versions earlier than 1.9 must not claim power policy ownership. With these earlier versions of WDF, USB selective suspend works properly only if WinUSB.sys is the PPO.

### I/O Queues in UMDF USB Drivers

For a UMDF driver that supports selective suspend, whether the UMDF driver owns power policy for its device determines the type of I/O queues that it can use. UMDF drivers that support selective suspend and are PPOs can use queues that are either power managed or not power managed. UMDF USB drivers that support selective suspend but are not the PPO should not use any power-managed I/O queues.

If an I/O request arrives for a power-managed queue while the device is suspended, the framework does not present the request unless the driver is PPO, as shown in Figure 1 earlier in this paper. If the UMDF driver is not the PPO for the device, the framework cannot power up the device on its behalf. As a result, the request remains stuck in the power-managed queue. The request never reaches WinUSB, so WinUSB cannot power up the device. Consequently, the device stack can stall.

If the queue is not power managed, the framework presents I/O requests to the UMDF driver even when the device is powered down. The UMDF driver formats the request and forwards it down the device stack to the default I/O target in the usual way. Special code is not required. When the request reaches the PPO (WinUSB.sys), WinUSB.sys powers up the device and performs the required I/O operation.

The sample driver in %WinDDK%\*BuildNumber*\Src\Usb\OsrUsbFx2\umdf\Fx2\_Driver\IdleWake defines the constant \_NOT\_POWER\_POLICY\_OWNER\_ when you build the non-PPO version of the driver. When the driver creates a queue for read and write requests, it determines whether to create a power-managed queue by checking for the constant.

To create the queue, the driver calls the driver-defined CMyQueue::Initialize method, which takes the following three parameters:

* *DispatchType*, a WDF\_IO\_QUEUE\_DISPATCH\_TYPE enumeration value that indicates how the queue dispatches requests.
* *Default*, a Boolean that indicates whether the queue is a default queue.
* *PowerManaged*, a Boolean that indicates whether the queue is power managed.

The following code snippet shows the driver’s call to the CMyQueue::Initialize method as part of read-write queue creation:

#if defined(\_NOT\_POWER\_POLICY\_OWNER\_)

 powerManaged = false;

#else

 powerManaged = true;

#endif

hr = \_\_super::Initialize(WdfIoQueueDispatchParallel,

 true,

 powerManaged,

 );

CMyQueue::Initialize then calls **IWDFDevice::CreateIoQueue** to create the queue as follows:

hr = m\_FxDevice->CreateIoQueue(

 callback,

 Default,

 DispatchType,

 PowerManaged,

 FALSE,

 &fxQueue

 );

This code sequence results in a default queue that dispatches requests in parallel. If the driver is the PPO the queue is power managed, and if the driver is not the PPO the queue is not power managed.

## Supporting USB Selective Suspend in a UMDF PPO

To support selective suspend, a UMDF USB driver that is the PPO for its device stack must do the following:

1. Claim power policy ownership for the device stack, typically in the **IDriverEntry::OnDeviceAdd**method on its driver callback object, as described earlier.

2. Enable selective suspend by calling the **IWDFDevice2::AssignS0IdleSettings** method on the framework device object.

To enable USB selective suspend from a PPO

Call **IWDFDevice2::AssignS0IdleSettings**, typically from the **OnPrepareHardware** method on the device callback object. Set the parameters to AssignS0IdleSettings as follows:

* *IdleCaps* to **IdleUsbSelectiveSuspend.**
* *DxState* to the device sleep state to which the framework transitions the idle device. For USB selective suspend, specify **PowerDeviceMaximum**, which indicates that the framework should use the value that the bus driver specified.
* *IdleTimeout* to the number of milliseconds that the device must be idle before the framework transitions it to *DxState.*
* *UserControlOfIdleSettings* to **IdleAllowUserControl** if your driver allows users to manage the idle settings, or otherwise to **IdleDoNotAllowUserControl**.
* *Enabled* to **WdfUseDefault** to enable selective suspend by default, but to allow the user’s setting to override the default.

The following example shows how the IdleWake\_PPO driver calls this method in its internal CMyDevice::SetPowerManagement method:

 hr = m\_FxDevice->AssignS0IdleSettings( IdleUsbSelectiveSuspend,

 PowerDeviceMaximum,

 IDLE\_TIMEOUT\_IN\_MSEC,

 IdleAllowUserControl,

 WdfUseDefault);

If the device hardware can generate a wake signal, the UMDF driver can also support system wake from S1, S2, or S3. For details, see “System Wake in a UMDF Driver” later in this paper.

## Supporting USB Selective Suspend in a non-PPO UMDF Driver

A UMDF function driver that is not the PPO can support selective suspend by using the features of the underlying WinUSB.sys driver. The UMDF driver must notify WinUSB that the device and driver support selective suspend and must enable selective suspend either in the INF file or by setting power policy on the USB target device object.

If a UMDF function driver enables selective suspend, the underlying WinUSB.sys driver determines when the device is idle. WinUSB starts an idle time-out counter when no transfers are pending or when the only pending transfers are IN transfers on an interrupt or bulk endpoint. By default, the idle time-out is 5 seconds, but the UMDF driver can change this default.

When WinUSB.sys determines that the device is idle, it sends a request to suspend the device down the kernel-mode device stack. The bus driver changes the state of the hardware as appropriate. If all device functions on the port have been suspended, the port enters the USB selective suspend state.

If an I/O request arrives at WinUSB.sys while the device is suspended, WinUSB.sys resumes device operation if the device must be powered up to service the request. The UMDF driver does not require any code to resume the device while the system remains in S0.

If the device hardware can generate a wake signal, the UMDF driver can also support system wake from S1, S2, or S3. For details, see “System Wake in a UMDF Driver” later in this paper.

A UMDF driver that is not the PPO can support selective suspend by taking the following two steps:

1. Notifying WinUSB.sys that the device and driver support selective suspend.

2. Enabling USB selective suspend.

In addition, the driver can optionally:

* Set a time-out value for the device.
* Allow the user to enable or disable selective suspend.

For an example of how to implement USB selective suspend in a UMDF USB function driver that is not the PPO, see the Fx2\_Driver sample in the WDK. This sample is located at %WinDDK%\*BuildNumber*\Src\Usb\OsrUsbFx2\Umdf\Fx2\_Driver\
IdleWake\_Non-PPO.

To notify WinUSB about selective suspend support

To notify WinUSB.sys that the device can support USB selective suspend, the device INF must add the **DeviceIdleEnabled** value to the device’s hardware key and set the value to 1. The following example shows how the Fx2\_Driver sample adds and sets this value in the WUDFOsrUsbFx2\_IdleWakeNon-PPO.Inx file:

[OsrUsb\_Device\_AddReg]

...

HKR,,"DeviceIdleEnabled",0x00010001,1

To enable USB selective suspend

A UMDF USB driver can enable USB selective suspend either at runtime or during installation in the INF.

* To enable support at runtime, the function driver calls **IWDFUsbTargetDevice::SetPowerPolicy** and sets the *PolicyType* parameter to AUTO\_SUSPEND and the *Value* parameter to TRUE or 1. The following example shows how the Fx2\_Driver sample enables selective suspend in the DeviceNonPpo.cpp file:

BOOL AutoSuspend = TRUE;

hr = m\_pIUsbTargetDevice->SetPowerPolicy( AUTO\_SUSPEND,

 sizeof(BOOL),

 (PVOID) &AutoSuspend );

* To enable support during installation, the INF includes an **AddReg** directive that adds the **DefaultIdleState** value to the device’s hardware key and sets the value to 1. For example:

HKR,,"DefaultIdleState",0x00010001,1

To set an idle time-out value

By default, WinUSB suspends the device after 5 seconds if no transfers are pending or if the only pending transfers are IN transfers on an interrupt or bulk endpoint. A UMDF driver can change this idle time-out value either at installation in the INF or at runtime.

* To set an idle time-out at installation, the INF includes an **AddReg** directive that adds the **DefaultIdleTimeout** value to the device’s hardware key and sets the value to the time-out interval in milliseconds.

The following example sets the time-out to 7 seconds:

HKR,,"DefaultIdleTimeout",0x00010001,7000

* To set an idle time-out at runtime, the driver calls **IWDFUsbTargetDevice::SetPowerPolicy** with *PolicyType* set to SUSPEND\_DELAY and *Value* to the idle time-out value, in milliseconds. In the following example from the Device.cpp file, the Fx2\_Driver sample sets the time-out to 10 seconds:

HRESULT hr;

ULONG value;

value = 10 \* 1000;

hr = m\_pIUsbTargetDevice->SetPowerPolicy( SUSPEND\_DELAY,

 sizeof(ULONG),

 (PVOID) &value );

To provide user control of USB selective suspend

UMDF USB drivers that use WinUSB selective suspend support can optionally allow the user to enable or disable selective suspend. To do so, include an **AddReg** directive in the INF that adds the **UserSetDeviceIdleEnabled** value to the device’s hardware key and sets the value to 1.

The following shows the string to use for the **AddReg** directive:

HKR,,"UserSetDeviceIdleEnabled",0x00010001,1

If **UserSetDeviceIdleEnabled** is set, the device’s **Properties** dialog box includes a **Power Management** tab that allows the user to enable or disable USB selective suspend.

## System Wake in a UMDF Driver

In a UMDF driver, support for system wake is independent of support for selective suspend. A UMDF USB driver can support both system wake and selective suspend, neither system wake nor selective suspend, or either system wake or selective suspend. A device that supports system wake can wake the system from a sleep state (S1, S2, or S3).

A UMDF USB PPO driver can support system wake by providing wake-up information for the framework’s driver object. When an external event triggers system wake, the framework returns the device to the working state.

A USB non-PPO driver can use the system wake support that the WinUSB.sys driver implements.

To support system wake in a UMDF USB driver that is the PPO

Call the **IWDFDevice2::AssignSxWakeSettings** method on the framework’s device object with the following parameters:

* *DxState* to the power state to which the device transitions when the system enters a wakeable S*x* state. For USB devices, specify **PowerDeviceMaximum** to use the value that the bus driver specified.
* *UserControlOfWakeSettings* to **WakeAllowUserControl** if your driver allows users to manage the wake settings or otherwise to **WakeDoNotAllowUserControl**.
* *Enabled* to **WdfUseDefault** to enable wake by default, but to allow the user’s setting to override the default.

The following example shows how the IdleWake\_PPO driver calls this method in its internal CMyDevice::SetPowerManagement method:

hr = m\_FxDevice->AssignSxWakeSettings( PowerDeviceMaximum,

 WakeAllowUserControl,

 WdfUseDefault);

To enable system wake through WinUSB in a non-PPO Driver

To enable system wake through WinUSB, the driver’s INF adds the registry value **SystemWakeEnabled** to the device’s hardware key and sets it to 1. The IdleWake\_Non-PPO sample enables system wake as follows:

[OsrUsb\_Device\_AddReg]

...

HKR,,"SystemWakeEnabled",0x00010001,1

By setting this value, the driver both enables system wake and allows the user to control the ability of the device to wake the system. In Device Manager, the power management settings property page for the device includes a check box with which the user can enable or disable system wake.

# Selective Suspend in a KMDF Function Driver

If the USB driver requires features or resources that are not available in user mode, you should supply a KMDF function driver. KMDF drivers implement selective suspend by setting relevant values in a KMDF initialization structure and then supplying the appropriate callback functions. KMDF handles the details of communicating with lower drivers to suspend and resume the device.

## Guidelines for Selective Suspend in KMDF Drivers

KMDF drivers that support selective suspend must follow these guidelines:

* A KMDF function driver must be the PPO for its device stack. By default, KMDF function drivers are the PPO.
* A KMDF function driver that supports selective suspend can use queues that are power managed or queues that are not power managed. By default, queue objects for PPOs are power managed.

### Power Policy Ownership and KMDF USB Drivers

By default, the KMDF function driver for a USB device is the PPO for the device stack. KMDF manages selective suspend and resume on behalf of this driver.

### I/O Queue Configuration in KMDF Drivers

A KMDF function driver that supports selective suspend can use queues that are power managed or queues that are not power managed. Typically, a driver configures a queue that is not power managed to receive incoming device I/O control requests and configures one or more power-managed queues to receive read, write, and other power-dependent requests. When a request arrives at a power-managed queue, KMDF ensures that the device is in D0 before it presents the request to the driver.

If you are writing a KMDF filter driver that is layered above the PPO in the device stack, you must not use power-managed queues. The reason is the same as for UMDF drivers. The framework does not present requests from power-managed queues while the device is suspended, so the use of such queues could stall the device stack.

## Selective Suspend Mechanism for KMDF Function Drivers

KMDF handles most of the work that is required to support USB selective suspend. It keeps track of I/O activity, manages the idle timer, and sends the device I/O control requests that cause the parent driver (Usbhub.sys or Usbccgp.sys) to suspend and resume the device.

If a KMDF function driver supports selective suspend, KMDF tracks the I/O activity on all power-managed queues that each device object owns. The framework starts an idle timer whenever the I/O count reaches zero. The default time-out value is 5 seconds.

If an I/O request arrives at a power-managed queue that belongs to the device object before the idle time-out period expires, the framework cancels the idle timer and does not suspend the device.

When the idle timer expires, KMDF issues the requests that are required to put the USB device in the suspended state. If a function driver uses a continuous reader on a USB endpoint, the reader’s repeated polling does not count as activity toward the KMDF idle timer. However, in the*EvtDeviceD0Exit* callback function, the USB driver must manually stop the continuous reader and any other I/O targets that are fed by queues that are not power managed to ensure that the driver does not send I/O requests while the device is not in the working state. To stop the targets, the driver calls **WdfIoTargetStop** and specifies **WdfIoTargetWaitForSentIoToComplete** as the target action. In response, the framework stops the I/O target only after all I/O requests that are in the target’s I/O queue have been completed and any associated I/O completion callbacks have run.

By default, KMDF transitions the device out of D0 and into the device power state that the driver specified in the idle settings. As part of the transition, KMDF calls the driver’s power callback functions in the same way that it would for any other power-down sequence.

After the device has been suspended, the framework automatically resumes the device when any of the following events occur:

* An I/O request arrives for any of the driver’s power-managed queues.
* The user disables USB selective suspend by using Device Manager.
* The driver calls **WdfStopIdle**, asdescribed in “Preventing Device Suspension” later in this paper.

To resume the device, KMDF sends a power-up request down the device stack and then invokes the driver’s callback functions in the same way that it would for any other power-up sequence.

For detailed information about the callbacks that are involved in the power-down and power-up sequences, see “Plug and Play and Power Management in WDF Drivers” on the WHDC Web site.

## Supporting USB Selective Suspend in a KMDF Function Driver

To implement USB selective suspend in a KMDF function driver:

* Initialize power policy settings that are related to idle, including idle time-out.
* Optionally include logic to temporarily prevent suspension or resume operation when the driver determines that the device should not be suspended because of an open handle or other reason that is not related to the device’s I/O queues.
* In a USB driver for a human interface device (HID), indicate in the INF that it supports selective suspend.

The Osrusbfx2 sample driver that is provided with the WDK in the %WinDDK%\*BuildNumber*\Src\Usb\OsrUsbFx2\Kmdf\Sys\Final directory shows how to initialize and configure USB selective suspend.

### Initializing Power Policy Settings in a KMDF Function Driver

To configure support for USB selective suspend, a KMDF driver uses the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure. The driver must first initialize the structure and can then set fields that provide details about the capabilities of the driver and its device. Typically, the driver fills in this structure in its *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* function.

To initialize the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure

After the driver creates the device object, the driver uses the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS\_INIT function to initialize the structure. This function takes two arguments:

* A pointer to the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure to initialize.
* An enumeration value that indicates support for selective suspend. The driver should specify **IdleUsbSelectiveSuspend**.

If the driver specifies **IdleUsbSelectiveSuspend**, the function initializes the structure’s members as follows:

* **IdleTimeout** is set to **IdleTimeoutDefaultValue** (currently 5000 milliseconds or 5 seconds).
* **UserControlOfIdleSettings** is set to **IdleAllowUserControl.**
* **Enabled** is set to **WdfUseDefault**, which indicates that selective suspend is enabled but a user can disable it if the **UserControlOfIdleSetttings** member permits it.
* **DxState** is set to **PowerDeviceMaximum**, which uses the reported power capabilities for the device to determine the state to which to transition the idle device.

To configure USB selective suspend

After the driver initializes the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure, the driver can set other fields in the structure and then call **WdfDeviceAssignS0IdleSettings** to pass these settings to the framework. The following fields apply to USB function drivers:

* **IdleTimeout**—The interval, in milliseconds, that must elapse without receiving an I/O request before the framework considers the device idle. The driver can specify a ULONG value or can accept the default.
* **UserControlOfIdleSettings**—Whether the user can modify the device’s idle settings. Possible values are **IdleDoNotAllowUserControl** and **IdleAllowUserControl**.
* **DxState**—The device power state to which the framework suspends the device. Possible values are **PowerDeviceD1**, **PowerDeviceD2**, and **PowerDeviceD3**.

USB drivers should not change the initial setting of this value. The WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS\_INIT function sets this value to **PowerDeviceMaximum**, which ensures that the framework chooses the correct value based on the device capabilities.

The following code snippet is from the Osrusbfx2 sample driver’s Device.c file:

WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS idleSettings;

NTSTATUS status = STATUS\_SUCCESS;

//

// Initialize the idle policy structure.

//

WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS\_INIT(&idleSettings,
 IdleUsbSelectiveSuspend);

idleSettings.IdleTimeout = 10000; // 10 sec

status = WdfDeviceAssignS0IdleSettings(Device, &idleSettings);

if ( !NT\_SUCCESS(status)) {

 TraceEvents(TRACE\_LEVEL\_ERROR, DBG\_PNP,

 "WdfDeviceSetPowerPolicyS0IdlePolicy failed %x\n",
 status);

 return status;

}

In the example, the driver calls WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS\_INIT, specifying **IdleUsbSelectiveSuspend**. The driver sets **IdleTimeout** to 10,000 milliseconds (10 seconds) and accepts the framework defaults for **DxState** and **UserControlOfIdleSettings**. As a result, the framework transitions the device to the D3 state when it is idle and creates a Device Manager property page that allows users with administrator privilege to enable or disable device idle support. The driver then calls **WdfDeviceAssignS0IdleSettings** to enable idle support and register these settings with the framework.

A driver can call **WdfDeviceAssignS0IdleSettings** any time after it creates the device object. Although most drivers call this method initially from the *EvtDriverDeviceAdd* callback, this might not always be possible or even desirable. If a driver supports multiple devices or device versions, the driver might not know all device capabilities until it queries the hardware. Such drivers can postpone calling **WdfDeviceAssignS0IdleSettings** until the *EvtDevicePrepareHardware* callback.

At any time after its initial call to **WdfDeviceAssignS0IdleSettings**,the driver can change the idle time-out value and the device state in which the device idles. To change one or more settings, the driver simply initializes another WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure as described earlier and calls **WdfDeviceAssignS0IdleSettings** again.

### Preventing USB Device Suspension

Sometimes, a USB device should not be powered down even if no I/O requests are present within the time-out period—typically when a handle is open to the device or the device is charging. A USB driver can prevent the framework from suspending an idle device in such situations by calling **WdfDeviceStopIdle** and calling **WdfDeviceResumeIdle** when it is again acceptable for the device to be suspended.

**WdfDeviceStopIdle** stops the idle timer. If the **IdleTimeout** period has not expired and the device has not yet been suspended, the framework cancels the idle timer and does not suspend the device. If the device has already been suspended, the framework returns the device to the working state. **WdfDeviceStopIdle** does not prevent the framework from suspending the device when the system changes to an S*x* sleep state. Its only effect is to prevent device suspension while the system is in the S0 working state. **WdfDeviceResumeIdle** restarts the idle timer.

These two methods manage a reference count on the device, so if the driver calls **WdfDeviceStopIdle** several times, the framework does not suspend the device until the driver has called **WdfDeviceResumeIdle** the same number of times. A driver must not call **WdfDeviceResumeIdle** without first calling **WdfDeviceStopIdle.**

### Including a Registry Key (HID Drivers Only)

KMDF upper filter drivers for USB HID devices must indicate in the INF that they support selective suspend so that the Microsoft-supplied HIDClass.sys port driver can enable selective suspend for the HID stack. The INF should include an **AddReg** directive that adds the **SelectiveSuspendEnabled** key and set its value to 1, as the following string shows:

HKR,,"SelectiveSuspendEnabled",0x00000001,0x1

For an example, see Hidusbfx2.inx in the WDK at %WinDDK%\*BuildNumber*\Src\Hid\
Hidusbfx2\sys.

## Remote Wake Support for KMDF Drivers

As with selective suspend, KMDF incorporates support for wakeup, so that a USB device can trigger a wake signal while the device is idle and the system is in the working state (S0) or in a sleep state (S1–S4). In KMDF terms, these two features are called “wake from S0” and “wake from S*x,*” respectively.

For USB devices, wakeup merely indicates that the device itself can initiate the transition from a lower-power state to the working state. Thus, in USB terms, wake from S0 and wake from S*x* are the same, and are called “remote wake.”

KMDF USB function drivers do not require any code to support wake from S0 because KMDF provides this capability as part of the selective suspend mechanism. However, to support remote wake when the system is in S*x*, a function driver must:

* Check whether the device supports remote wake by calling **WdfUsbTargetDeviceRetrieveInformation**.
* Enable remote wake by initializing wake settings and calling **WdfDeviceAssignSxWakeSettings.**

KMDF drivers typically configure wake support at the same time that they configure support for USB selective suspend in the *EvtDriverDeviceAdd* or *EvtDevicePrepareHardware* function.

### Checking Device Capabilities

Before a KMDF USB function driver initializes its power policy settings for idle and wake, it should verify that the device supports remote wake. To get information about device hardware features, the driver initializes a WDF\_USB\_DEVICE\_INFORMATION structure and calls **WdfUsbTargetDeviceRetrieveInformation**, typically in its *EvtDriverDeviceAdd* cor *EvtDevicePrepareHardware* callback.

In the call to **WdfUsbTargetDeviceRetrieveInformation**, the driver passes a handle to the device object and a pointer to the initialized WDF\_USB\_DEVICE\_INFORMATION structure. Upon successful return from the function, the **Traits** field of the structure contains flags that indicate whether the device is self powered, can operate at high speed, and supports remote wake.

The following example from the Osrusbfx2 KMDF sample shows how to call this method to determine whether a device supports remote wake. After these lines of code have run, the waitWakeEnable variable contains TRUE if the device supports remote wake and FALSE if it does not:

 WDF\_USB\_DEVICE\_INFORMATION deviceInfo;

// Retrieve USBD version information, port driver capabilities and device

// capabilites such as speed, power, etc.

//

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

status = WdfUsbTargetDeviceRetrieveInformation(

 pDeviceContext->UsbDevice,

 &deviceInfo);

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

### Enabling Remote Wakeup

In USB terminology, a USB device is enabled for remote wakeup when its DEVICE\_REMOTE\_WAKEUP feature is set. According to the USB specification, host software must set the remote wakeup feature on a device “only just prior” to putting the device to sleep. The KMDF function driver is required only to initialize the wake settings. KMDF and the Microsoft-supplied USB bus drivers issue the I/O requests and handle the hardware manipulation that is required to enable remote wakeup.

To initialize wake settings

1. Call WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS\_INIT to initialize a WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS structure.

This function sets the structure's **Enabled** member to **WdfUseDefault**, sets the **DxState** member to **PowerDeviceMaximum**, and sets the **UserControlOfWakeSettings** member to **WakeAllowUserControl**.

2. Call **WdfDeviceAssignSxWakeSettings** with the initialized structure.

As a result, the device is enabled to wake from the D3 state and the user can enable or disable the wake signal from the device property page in Device Manager.

The following code snippet from the Osrusbfx2 sample shows how to initialize wake settings to their default values:

WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS wakeSettings;

WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS\_INIT(&wakeSettings);

status = WdfDeviceAssignSxWakeSettings(Device, &wakeSettings);

if (!NT\_SUCCESS(status)) {

 return status;

}

For USB devices that support selective suspend, the underlying bus driver prepares the device hardware to wake. Consequently, USB function drivers rarely require an *EvtDeviceArmWakeFromS0* callback. The framework sends a selective suspend request to the USB bus driver when the idle time-out expires.

For the same reason, USB function drivers rarely require a *EvtDeviceWakeFromS0Triggered* or *EvtDeviceWakeFromSxTriggered* callback. Instead, the framework and the underlying bus driver handle all requirements for returning the device to the working state.

# Resources

#### WHDC Web site

Windows Driver Foundation (WDF)

<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

Plug and Play - Architecture and Driver Support

<http://www.microsoft.com/whdc/system/pnppwr/pnp/default.mspx>

#### White Papers on the WHDC Web site

Plug and Play and Power Management in WDF Drivers

<http://www.microsoft.com/whdc/driver/wdf/WDF_pnpPower.mspx>

When WDF Drivers Can Use Power-Managed I/O Queues

<http://www.microsoft.com/whdc/driver/tips/pow-mnged_queues.mspx>

Writing USB Drivers with WDF

<http://www.microsoft.com/whdc/driver/wdf/USB_WDF.mspx>

#### WDK documentation

PnP and Power Management in Framework-Based Drivers

<http://msdn.microsoft.com/en-us/library/aa490107.aspx>

PnP and Power Management in UMDF-based Drivers

<http://msdn.microsoft.com/en-us/library/aa511013.aspx>

USB Power Management

<http://msdn.microsoft.com/en-gb/library/ms793249.aspx>

#### Books

*Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith

<http://www.microsoft.com/MSPress/books/10512.aspx>