Plug and Play and Power Management in WDF Drivers

December 11, 2009

Abstract

The Windows® Driver Foundation (WDF) implements a fully integrated model for Plug and Play and power management in both the user-mode driver framework (UMDF) and the kernel-mode driver framework (KMDF). The model provides intelligent defaults so that some drivers do not require any code to support simple Plug and Play or power management. To support more complex features, drivers implement event callbacks. This paper provides guidelines for implementing Plug and Play and power management support in UMDF and KMDF drivers.

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 (KMDF only)

The current version of this paper is maintained on the Web at:   
 <http://www.microsoft.com/whdc/driver/wdf/WDF_pnpPower.mspx>

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>.

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 |  |  |  |
| December 11, 2009 | Added information about new UMDF features; updated links and sample locations for Windows 7 release | | | |
| April 9, 2007 | First publication | | | |

# Contents

[Contents 3](#_Toc248298183)

[Introduction 4](#_Toc248298184)

[About Plug and Play 5](#_Toc248298185)

[About Power States 6](#_Toc248298186)

[About Power Policy 7](#_Toc248298187)

[Plug and Play and Power Management Support in WDF 8](#_Toc248298188)

[Plug and Play and Power Management Defaults 8](#_Toc248298189)

[I/O Queues and Power Management 8](#_Toc248298190)

[Plug and Play and Power Event Callbacks 8](#_Toc248298191)

[Idle and Wake Support 10](#_Toc248298192)

[Power-Pageable and Non-Power-Pageable Drivers 11](#_Toc248298193)

[Callback Sequences for Plug and Play and Power Management 12](#_Toc248298194)

[Device Enumeration and Startup 16](#_Toc248298195)

[Device Power-Down and Removal 19](#_Toc248298196)

[Surprise Removal 22](#_Toc248298197)

[UMDF Surprise-Removal Sequence 23](#_Toc248298198)

[KMDF Surprise-Removal Sequence 23](#_Toc248298199)

[How to Implement Plug and Play and Power Management in WDF Drivers 24](#_Toc248298200)

[Plug and Play and Power Management in Software-Only Drivers 25](#_Toc248298201)

[UMDF Example: Plug and Play in a Software-Only Filter Driver 26](#_Toc248298202)

[KMDF Example: Plug and Play in a Software-Only Filter Driver 27](#_Toc248298203)

[Framework Actions for Software-Only Drivers 28](#_Toc248298204)

[Plug and Play and Power Management in Simple Hardware Drivers 29](#_Toc248298205)

[Device Power-Up Initialization and Power-Down Teardown 29](#_Toc248298206)

[Power Management for Queues in Hardware Function Drivers 30](#_Toc248298207)

[UMDF Example: Plug and Play and Power Code in a Protocol Function Driver 31](#_Toc248298208)

[Power-Managed Queue for a UMDF Driver 32](#_Toc248298209)

[IPnpCallbackHardware Methods 32](#_Toc248298210)

[IPnpCallback Methods 33](#_Toc248298211)

[KMDF Example: Plug and Play and Power Code in a Simple Hardware   
Function Driver 34](#_Toc248298212)

[KMDF Example: Register Callbacks and Set Up Power-Managed Queues 34](#_Toc248298213)

[KMDF Example: D0 Entry and D0 Exit Callbacks 36](#_Toc248298214)

[Framework Actions for a Simple Hardware Function Driver 37](#_Toc248298215)

[Advanced Power Management 39](#_Toc248298216)

[Device Power-Down Idle Support 39](#_Toc248298217)

[Idle Settings and Management 40](#_Toc248298218)

[How to Choose Idle Times and Idle States 42](#_Toc248298219)

[Device Wake Support 43](#_Toc248298220)

[How to Implement Wake from Sx 44](#_Toc248298221)

[How to Implement Wake from S0 46](#_Toc248298222)

[KMDF Example: Support for Device Idle and Wake 48](#_Toc248298223)

[Framework Actions Supporting Device Idle 49](#_Toc248298224)

[Framework Actions Supporting Device Wake 50](#_Toc248298225)

[Resources 51](#_Toc248298226)

# Introduction

Plug and Play and power management encompass a variety of activities that are involved in the installation, configuration, and operation of devices. The following are just a few situations that require Plug and Play or power management support:

* The user connects a new MP3 player to a running system.
* The user unexpectedly removes a USB flash drive.
* While the system is running, the user plugs in an Ethernet cable to connect the computer to a network.
* While the system is suspended, the user wakes it up by moving the mouse.
* The administrator configures the system to hibernate after it is idle for an extended period.

To properly support Plug and Play and power management, the operating system, drivers, system administration software, device installation software, system hardware, and device hardware must all work together.

Windows® and the Windows Driver Model (WDM) expose a complicated model for Plug and Play and power management that depends on the driver to track both the state of its device and the state of the system, thus in effect implementing its own informal state machine. WDM drivers must know which Plug and Play and power requests to handle in each state and which operations to perform in response to those requests. Some requests require the driver to perform one operation if the device is powered up and a different operation if it is not. Other requests require no driver action at all, but the driver must nevertheless include code that parses the request and checks the current device and system state to determine whether action is required.

In contrast, the user-mode driver framework (UMDF) and the kernel-mode driver framework (KMDF) that are part of the Windows Driver Foundation (WDF) implement intelligent default behavior and expose a set of state-specific callbacks that drivers can implement to customize the Plug and Play and power behavior. WDF tracks the state of the device and the state of the system and maintains information about the Plug and Play and power capabilities of the driver and device hardware. The frameworks can also manage a driver’s I/O queues with respect to the device’s power state. If an I/O request arrives while the device is not powered on, the frameworks can power up the device.

A WDF driver can “opt in” to device-specific handling for more complicated situations by implementing callbacks for such events as setup initialization, shutdown cleanup, power on, power off, and so forth. The WDF defaults apply to any event for which the driver does not implement a callback.

WDF provides a wide range of Plug and Play and power management options for drivers. For example:

* By default, both KMDF and UMDF drivers support the fundamental Plug and Play and power management features, including fast resume and suspend.

A driver requires code only to save and restore device context and, in some KMDF drivers, to enable and disable device interrupts.

* WDF drivers can support automatically suspending an idle device on a running system.

During idle periods, the framework puts the device in a low-power state. For USB devices, the driver can use selective suspend. To support this extra functionality, most drivers require only one additional function call.

* WDF drivers can support a device wake signal that wakes the device or the system.

The driver identifies the power states at which the device can trigger a wake signal. A WDF driver can support wake at any system power state other than the off state. Implementing wake typically requires adding just a few callback functions to a driver.

In each of these examples, a driver supplies only the code that is required to manipulate its device. The framework tracks device and system state and calls the driver at its registered callbacks to perform device-specific actions.

## About Plug and Play

Plug and Play is a combination of system software, device hardware, and device driver support through which a computer system can recognize and adapt to hardware configuration changes with little or no intervention by an end user. An end user can add devices to and remove devices from the system without doing technical, hardware-level configuration. For example, a user can plug in a USB flash drive to transfer files or can dock a portable computer and use the docking station’s keyboard, mouse, and monitor without making manual configuration changes.

If the device hardware and driver support Plug and Play, Windows recognizes the new devices, loads the proper drivers, and starts the devices to make them available for the user. The PnP manager is the kernel subsystem that recognizes hardware during initial system installation, recognizes hardware changes that occur between system boots, and responds to hardware events such as docking or undocking and device insertion or removal.

The bus driver detects and enumerates devices and requests resources for those devices. The PnP manager gathers resource requests from all bus drivers and assigns resources to the devices. Resources are not dynamically configurable for legacy devices, so the PnP manager assigns resources to legacy devices first. If the user adds a new device that requires resources that are already in use, the PnP manager reconfigures resource assignments.

At device object (DO) initialization, the bus driver indicates which of the following Plug and Play features its device supports:

* The device can be ejected from its slot.
* The device is a docking station.
* The device can be removed while the system is running.
* A user can remove the device without using the Unplug or Eject Hardware application.
* The device can be locked in its slot to prevent ejection.
* The device can be hidden in Device Manager.

## About Power States

A power state describes the level of power consumption for the system or for an individual device. System power states are named S0 for the working state and S*x*, where *x* is a state number between 1 and 5, for the sleep states. Device power states are named D0 and D*x*, where *x* is a state number between 1 and 3. The state number is inversely related to power consumption: higher numbered states use less power.

The term “highest-powered state” means the state that uses the most power, and conversely for “lowest-powered state.” Therefore, D0 is a higher-powered state than D1, and D3 is a lower-powered state than D2.

For the system, state S0 is the highest powered, most functional, fully-on working state. State S4 is the hibernation state. State S5 is the off state. States S1, S2, and S3 are sleep states, with progressively lower levels of power consumption.

For a device, D0 is the fully-powered working state and D3 is the powered-down (off) state. All devices must support these two states. The exact definitions of the intermediate power states are bus and device specific, unlike the system states where S4 and S5 are universally defined. Not all devices support intermediate power states. Many devices support only the D0 and D3 states.

**Note** For PCI devices, the PCI specification defines the D3hot and D3cold states. In Windows, D3hot means that the device is in D3 and its parent bus is in D0. D3cold means that the device is in D3 and its parent bus is in D*x*.

A device can transition from D0 to any lower-power state (D*x*) and from any lower-power state to D0. However, a device cannot transition from one sleep state (D*x*) to another; it must return to D0 before it can enter a different sleep state. In addition, the device must be in D0 when the driver arms or disarms the wake signal. The reason is that access to the device hardware is prohibited when the device is in a sleep state, so the driver must return the device to the working state before it performs any hardware-related activity.

The power state of a device is related to the system state but is not required to match it. For example, many devices can be in the off state (D3) when the system is in the working state (S0). Usually, a device’s power state is no higher than that of the system because many devices get their power from the system. Devices that are enabled to wake a suspended system are exceptions; such devices are typically in a sleep state (D1 or D2) when the system is in a sleep state.

#### UMDF Drivers and Device Power Capabilities

For UMDF drivers, the underlying bus driver sets the power capabilities of the device and the UMDF driver cannot change them.

#### KMDF Drivers and Device Power Capabilities

A KMDF driver can set power capabilities in the same way that it sets Plug and Play capabilities. Typically a bus driver sets the capabilities of the devices that are attached to the bus, but a function or filter driver can override the bus driver’s settings. The settings include the following:

* The power states that the device supports in addition to D0 and D3.
* The power states from which the device can respond to a wake signal.
* The highest-powered D*x* state that the device supports for each system S*x* state.
* The highest-powered D*x* state from which the device can trigger a wake signal to the system.
* The lowest-powered S*x* state at which the device can trigger a wake signal.
* Approximate latency time for the device to return to D0 from each sleeping state.
* The “ideal” D*x* state that the device should enter when its wake signal is not enabled and the system is entering a sleep state.

When the system enters an S*x* state, the framework transitions the device to either the ideal D*x* state or the lowest-powered D*x* state that the device can support at the given S*x* state—whichever is higher powered.

The driver reports the power capabilities to the framework, and the framework, in turn, reports them to the system. The drivers in the device stack cooperatively set power capabilities, so it is possible for another driver that is higher in the stack to override the values that a KMDF driver sets.

## About Power Policy

The power policy for a device determines which power state the device should be in at any given time. One driver in each device stack is responsible for controlling the device’s power policy, and that driver is called the power policy owner (PPO) for the device stack:

* A UMDF driver must explicitly indicate that it is the PPO.
* KMDF assumes, by default, that the functional device object (FDO) is the PPO for the device. If the device is controlled by a raw physical device object (PDO), KMDF assumes that the raw PDO is the PPO.

The function driver is responsible for the device’s functional operation and is therefore most likely to have the necessary information about the best way to manage device power. A KMDF filter driver can indicate that it is the PPO for its device by notifying the framework during initialization.

The PPO is not necessarily the driver that manipulates the device hardware to change the power state. It is simply the driver that specifies when the device power state transitions should occur.

Drivers that claim power policy ownership must ensure, through some means outside standard operating system control, that they are indeed the only PPO in their stack. Usually, this mechanism is a documented policy that indicates which driver is the power policy manager.

# Plug and Play and Power Management Support in WDF

WDF implements Plug and Play and power management with several internal state machines. Both KMDF and UMDF use the same state machines. An event is associated with the specific actions that a driver might be required to perform at a particular time, and the driver implements the event callbacks to perform the actions that its device requires. The callbacks are called in a defined order and each conforms to a “contract,” so that both the device and the system are guaranteed to be in a particular state when the driver is called to perform an action.

## Plug and Play and Power Management Defaults

Although WDF provides great flexibility so that a driver can control detailed aspects of its device’s Plug and Play capabilities, WDF also implements defaults that enable many filter drivers and software-only drivers to omit any Plug and Play code whatsoever. By default, WDF supports all of the Plug and Play features that such drivers need.

By default, WDF assumes the following:

* The device supports D0 and D3.
* The device and driver do not support idle or wake.
* The I/O queues for an FDO or a PDO are power managed.

## I/O Queues and Power Management

The frameworks implement power management for I/O queues, so that the queue automatically starts and stops when the device enters and leaves the working state. Such a queue is “power managed.” The framework dispatches I/O requests from a power-managed queue to the driver only when the device hardware is accessible and in the working power state. The driver is not required to maintain device state or to check device state each time it receives an I/O request from a power-managed queue.

By default, the I/O queues of KMDF FDOs and PDOs are power managed. A driver can easily change this default to create a non-power-managed queue or to configure power-managed queues for a filter DO. If an I/O request arrives while the device is in a low-power idle state, the framework can restore device power before it delivers the request if the driver is the PPO or is layered below the PPO in the device stack. However, a driver should not use power-managed queues if it is layered above the PPO in the device stack. Doing so can stall the device stack.

## Plug and Play and Power Event Callbacks

Most of the Plug and Play and power callbacks are defined in pairs: one event occurs upon entry to a state, and the other occurs upon exit from the state. Generally, one member of the pair performs a task that the other reverses. A driver can implement one, both, or neither of a pair. In a UMDF driver where both methods are defined on a single interface, the driver must implement the entire interface on the device callback object but can supply minimal implementations of the methods that it does not require.

The frameworks are designed to work with drivers on an opt-in basis. A driver implements callbacks for only the events that affect its device. For example, some drivers must save device state immediately before the device leaves the D0 power state and must restore device state immediately after the device reenters the D0 power state. As another example, a device might have a motor or fan that the driver must start when the device enters D0 and must stop before the device leaves D0. A driver can implement callback functions that are invoked at those times. If the device does not require service at those particular times, its driver does not implement the callbacks.

Table 1 summarizes the types of Plug and Play and power features that a driver might require and the UMDF interfaces and KMDF event callbacks that the driver implements to support those features.

Table 1. Plug and Play and Power Callbacks for WDF Drivers

|  |  |  |
| --- | --- | --- |
| If your driver… | Implement this UMDF interface and its methods on the device callback object… | Implement this KMDF event callback … |
| Uses self-managed I/O | **IPnpCallbackSelfManagedIo::**  **Xxx** | *EvtDeviceSelfManagedIoXxx* |
| Requires service immediately before the device is initially powered up and after it powers down during resource rebalancing or device removal | **IPnpCallbackHardware::**  **OnPrepareHardware**  **OnReleaseHardware** | *EvtDevicePrepareHardware* and *EvtDeviceReleaseHardware* |
| Requires service immediately after the device enters D0 and before it leaves D0 | **IPnpCallback::**  **OnD0Entry**  **OnD0Exit** | *EvtDeviceD0Entry* and *EvtDeviceD0Exit* |
| Requires the opportunity to evaluate and veto each attempt to stop or remove the device | **IPnpCallback::**  **OnQueryStop**  **OnQueryRemove** | *EvtDeviceQueryStop and EvtDeviceQueryRemove* |
| Requires additional service at surprise-removal beyond the normal device removal processing | **IPnpCallback::**  **OnSurpriseRemoval** | *EvtDeviceSurpriseRemoval* |
| Supports a device that can be powered down when idle and wake itself while the system remains in the working state | **IPowerPolicyCallbackWakeFromS0::**  **OnArmWakeFromS0**  **OnDisarmWakeFromS0**  **OnWakeFromS0Triggered** | *EvtDeviceArmWakeFromS0* and *EvtDeviceDisarmWakeFromS0*  *EvtDeviceWakeFromS0Triggered*  *EvtDeviceEnableWakeAtBus* and *EvtDeviceDisableWakeAtBus* |
| Supports a device that can wake the system | **IPowerPolicyCallbackWakeFromSx::**  **OnArmWakeFromSx**  **OnDisarmWakeFromSx**  **OnWakeFromSxTriggered** | *EvtDeviceArmWakeFromSx* and *EvtDeviceDisarmWakeFromSx EvtDeviceWakeFromSxTriggered*  *EvtDeviceEnableWakeAtBus* and *EvtDeviceDisableWakeAtBus* |

If you are familiar with WDM drivers, you probably remember that any time the system power state changes, the WDM PPO must determine the correct power state for its device and then send power management requests to put the device in that state at the appropriate time. The WDF state machine automatically translates system power events to device power events and notifies the driver to do the following:

* Transition the device to low power when the system transitions to S*x*.
* Return the device to full power when the system returns to S0.
* Enable the device’s wake signal so that it can be triggered while the device is in a D*x* state and the system is in the working state.
* Enable the device’s wake signal so that it can be triggered while the device is in a D*x* state and the system is in a sleep state.

WDF automatically provides for the correct behavior in device parent/child relationships for bus drivers that are the PPO for their device stack. If both a parent bus and a child device are powered down, WDF ensures that the parent bus is powered up before it transitions the child to the D0 state. Similarly, WDF ensures that the parent does not power down until all children have powered down. However, if a filter driver that is not the PPO enumerates children for a virtual device, the child device can power up regardless of the power state of the parent. Because the child is a virtual device, its power state is not dependent on that of the parent.

Because KMDF drivers have greater access to device hardware than UMDF drivers do, KMDF supports additional features such as hardware resource requirements and interrupts. Table 2 lists additional callbacks that apply only to KMDF drivers.

Table 2. Additional KMDF Plug and Play Callbacks

| If your driver… | Implement this KMDF event callback… |
| --- | --- |
| Manages device resource requirements | *EvtDeviceResourceRequirementsQuery EvtDeviceResourcesQuery EvtDeviceRemoveAddedResources EvtDeviceFilterAddResourceRequirements EvtDeviceFilterRemoveResourceRequirements* |
| Performs hardware-related tasks around interrupts | *EvtInterruptEnable* and *EvtInterruptDisable*  *EvtDeviceD0EntryPostInterruptsEnabled* and *EvtDeviceD0ExitPreInterruptsDisabled* |

## Idle and Wake Support

To manage idle devices, the framework notifies the driver to transition the device from the working state to the designated low-power state when the device is idle and to return the device to the working state when requests need to be processed. The driver supplies callbacks that initialize and deinitialize the device, save and restore device state, and enable and disable the device wake signal.

By default, a user who has the appropriate privileges can control both the behavior of the device while it is idle and the ability of the device to wake the system. WDF implements the required Windows Management Instrumentation (WMI) provider, and Device Manager displays a property page through which the user can configure the settings. The PPO for the device can disable this feature by specifying the appropriate enumeration value when it initializes certain power policy settings.

## Power-Pageable and Non-Power-Pageable Drivers

Most devices can be powered down without affecting the system’s ability to access the paging file or to write a hibernation file. The drivers for such devices are considered “power pageable”:

* All UMDF drivers are power pageable.
* Most KMDF drivers are power pageable.

A device that is in the hibernation path, however, must remain in D0 during some power transitions so that the system can write the hibernation file. A device that is in the paging path remains in D0 until the system has written the hibernation file, at which point the entire machine shuts off. The device stacks for the hibernation and paging devices are considered non-power pageable. A KMDF driver indicates that it can support the paging, hibernation file, or system dump file by calling **WdfDeviceSetSpecialFileSupport** and providing a callback for notification if the device is actually used for such a file.

For example, drivers in the video and storage stacks are non-power pageable because the system uses these devices during power-down. The monitor must remain on so that Windows can display information to the user. During transitions to S4, the target disk for the hibernation file and the disk that contains the paging file must remain in D0 so that the system can write the hibernation file. For the disks to retain power, every device on which they depend must also retain power—such as the disk controller, the PCI bus, the interrupt controller, and so on. All drivers in these device stacks must therefore be non-power pageable.

Most drivers should use the framework’s defaults, which are as follows:

* FDOs by default are power pageable.
* PDOs by default inherit the setting of the driver that enumerated them.

If the PDO is power pageable, all DOs that are attached to it must also be power pageable. For this reason, a bus driver typically marks its FDO as non-power pageable so that its PDOs inherit the same attribute. The DOs that load above the PDO can then be either power pageable or non-power pageable.

* Filter DOs use the same setting as the next lower driver in the stack. A driver cannot change the setting for a filter DO.

If the default is inappropriate, a function or bus driver can explicitly call the **WdfDeviceInitSetPowerPageable** or **WdfDeviceInitSetPowerNotPageable** method during DO initialization to change the default. These methods set and clear the DO\_POWER\_PAGABLE value in the **Flags** field of the underlying WDM DO for an FDO or PDO, but have no effect for filter DOs.

The framework can change the value of the DO\_POWER\_PAGABLE flag for any DO if the system notifies the driver that the device is used for a hibernation, paging, or dump file.

If you are certain that none of the drivers in the device stack must be non-power pageable, your driver can call **WdfDeviceInitSetPowerPageable**. This might be the case if you wrote all the drivers in the stack or if the requirements for the device stack are clearly documented. A PDO must not be power pageable unless the device stacks of all its child devices are also power pageable.

KMDF provides the following special handling for drivers that are non-power pageable:

* The framework disables—but does not disconnect—the device’s interrupt when the device leaves the D0 state. The framework cannot disconnect the interrupt because the required **IoDisconnectInterruptXxx** system call is pageable.
* The framework implements a watchdog timer on all callbacks for power and wake events. If the driver causes paging I/O after the paging file’s device has left D0, a deadlock occurs, thus hanging the system. When the timer expires, the system crashes so that the user can determine which driver caused the deadlock. You can use the **!wdfextendwatchdog** debugger extension to extend the time-out during debugging. KMDF does not provide a way to extend the time-out programmatically.
* A driver can determine whether it is currently in a nonpageable power state by calling **WdfDevStateIsNP**(**WdfDeviceGetDevicePowerState**()) from within a power or power policy callback function.

**WdfDeviceGetDevicePowerState** returns an enumeration value of the WDF\_DEVICE\_POWER\_STATE type, which identifies the detailed state of the framework’s state machine. For example, **WdfDevStatePowerD0** and **WdfDevStatePowerD0NP** are two distinct values that represent the pageable D0 state and the nonpageable D0 state, respectively.

**WdfDevStateIsNP** returns TRUE if the driver is currently in a nonpageable power state and FALSE otherwise. This value is valid only while the current callback function is running. After the callback returns, the power state can change. Therefore, if the driver must perform actions that involve paging, the driver should do so immediately upon determining that the device power state permits these actions.

# Callback Sequences for Plug and Play and Power Management

Plug and Play and power management handle the activities that are required to bring a newly inserted device to full operation and to remove an operational device from the system. When the user plugs in a new device, the system must determine the type and capabilities of the device, assign resources to the device, work with the device’s drivers to power up and initialize the device, and do whatever else is required to ready the device for operation. When the user unplugs the device, the system reverses this process. The core activities that are related to device arrival and startup follow a fixed sequence, as do the core activities that are related to shutdown and removal.

At each point in the sequence, the device stack is in a well-defined state. The WDF state machines track the device stack through the transitions from one state to another, and WDF defines callbacks that correspond to many of the state transitions. When such a state change occurs, the framework invokes the callback, if any, that applies to the new state.

For example, a key activity in the startup sequence is to initialize the device and driver when the device powers up. In most device stacks, the bus driver is responsible for ensuring that the device has power, but the function driver handles initialization. Depending on the type of device, the driver might perform initialization before power is applied, after power is applied, or both. WDF defines callbacks for each of these states. If the device generates interrupts, the driver can further request a callback after power is applied but before the interrupt is connected or immediately after the interrupt is connected.

A device stack can change state for numerous reasons. The following are among the most common:

* The device is added to the system.
* The device is removed from the system, either in an orderly way or by surprise.
* The system is powering up.
* The system is either hibernating or standing by.
* The system is shutting down.
* An idle device is being powered down to conserve energy.
* An idle device is being re-powered because I/O has arrived for it or because an external wake signal was triggered.
* Another device has been added to the system, and Windows consequently must rebalance resources.
* A driver is being upgraded or reinstalled.

The reason for the change determines exactly which callbacks the framework calls. For example, if the system is shutting down, the framework calls all the driver callbacks that are involved in stopping device I/O and powering down the device. However, if an idle device is powering down, only the callbacks to stop some device I/O and power down the device are required. The framework retains the DO and the device’s resources for use when the device resumes operation.

From the perspective of the driver, however, the reason for the state change is not important. The driver’s callbacks perform discrete tasks at clearly defined times—such as initializing the device after the hardware is powered up. Whether the device is powering up as part of initial installation or because it triggered a wake signal is irrelevant to the driver. For the driver, the important point is that the framework always invokes the callbacks in the same sequence and according to the contract for that state.

#### State Changes and the Callback Sequence

The framework invokes the relevant callbacks in a fixed sequence as the device is inserted and made fully operational and in the reverse order as the device is powered down and removed. However, devices are not often installed and removed. More typically, the system transitions to an S*x* state when the user closes the lid of a laptop or the driver for a network card transitions its device to a D*x* state when the cable is unplugged. In these situations, the entire sequence of callbacks is unnecessary. Instead, the framework calls only the part of the sequence that is required to put the device in the desired state.

During power-down, the framework stops the sequence at one of four points, depending on whether the device is:

* Transitioning to a lower-power state.
* Stopping to rebalance resources.
* Disabled or removed but still physically present—for example, if the user disabled the device in Device Manager.
* Physically removed.

With each subsequent callback, and at each subsequent stopping point, less device and driver functionality is available. When the framework returns the device to the working state, it starts the sequence of callbacks at the same point at which it stopped, but in the reverse order. If the device is removed unexpectedly (that is, “surprise removed”), the sequence differs somewhat, as described in “Surprise Removal” later in this paper.

After the device has left the working state and transitioned to a lower-powered state, the framework always returns the device to the working state before it changes the device power state again. For example, assume that a device is idling in state D2 when the user shuts down the system. The framework resumes the shutdown callback sequence at the stopping point and calls the callbacks in the reverse order to return the device to D0. Then it starts the sequence again, beginning with the device in the operational state, to perform the activities that are required for shutdown. Although it might seem counterintuitive, this approach ensures that the drivers for the device can perform any additional tasks that are required before the system shuts down. A driver might save less state information when it transitions to D2 than it requires for D3, so the interim transition to D0 lets the driver recover whatever additional information it requires before the device enters D3.

#### Framework Startup and Shutdown Sequences

To ensure that your driver implements the appropriate callbacks and that each one performs the right tasks, you must understand the order in which the framework invokes the callbacks.

Figure 1 lists the steps in the framework’s startup and shutdown sequences. The startup actions in the left column correspond to the shutdown actions in the right column. Not all steps apply to every type of driver or DO, and whether some steps apply depends on the reason for the startup or shutdown. For example, the framework does not call the driver to enable the device wake signal when the device is stopping so that the PnP manager can rebalance system resources. A surprise removal triggers the shutdown and removal sequence even though the hardware has already been physically removed.

The specific callbacks that the framework invokes to perform each of these actions are listed later in this paper.



Figure 1. Steps in framework startup and shutdown sequences

#### Callback Function Failures

If the callback at any of the steps returns a failure status, the framework tears down the device stack.

If the failure occurs during power-up, the framework calls the driver’s release-hardware callback—if the driver implements one—but does not call any other callbacks.

If the failure occurs during power-down, the framework continues to call the driver’s callbacks. Therefore, the callback methods in the power-down sequence must be able to tolerate failures that unresponsive hardware causes.

## Device Enumeration and Startup

Whenever Windows boots, the loader, the PnP manager, and the drivers cooperatively enumerate all devices on the system and build devnodes to represent them. The same procedure also occurs when a device is added to a running system, although on a smaller scale.

During device enumeration, the PnP manager loads the drivers and builds the device stack for each device that is attached to the system. To prepare a device for operation, the PnP manager sends a sequence of requests to each device stack to get the information about the resources that the device requires, the capabilities of the device, and so on. The drivers in the device stack respond to these requests one at a time from the bottom up, beginning with the driver that is closest to the hardware. Thus, the PDO powers on the device before the FDO receives a request to perform its power-on tasks. Power-down occurs in the opposite order. In short, the higher level drivers depend on the functionality of the drivers below them, so they start later and stop earlier.

The frameworks participate in this sequence on behalf of their drivers. The frameworks respond immediately to the PnP manager’s requests when they can and invoke driver callback functions to supply information and perform tasks for which driver input or actions are required. For framework drivers:

* All UMDF drivers that are not the PPO can implement the same set of callback interfaces.

The **IDriverEntry** interface is implemented on the driver callback object. The **IPnpCallback**, **IPnpCallbackHardware**, and **IPnpSelfManagedIo** interfaces are implemented on the device callback object. The **IQueueCallbackXxx** interfaces are implemented on the queue callback objects.

A UMDF driver that is PPO for its device stack can optionally implement the **IPowerPolicyCallbackWakeFromS0** and **IPowerPolicyCallbackWakeFromSx** interfaces on its device callback object.

* For KMDF drivers, most event callbacks apply to any type of DO.

However, the framework invokes some callbacks only for PDOs and other callbacks only for FDOs and filter DOs.

Figures 2 and 3 show the sequence of callbacks for UMDF drivers and for KMDF FDO and filter DOs that are involved in bringing a device to the fully operational state, starting from the Device Arrived state at the bottom of the figure.

In these figures, the broad horizontal lines mark the steps for starting a device. The column on the left side of the figure describes the step, and the column on the right lists the event callbacks that accomplish the step. If the device is stopped because the PnP manager is rebalancing system resources or if the device is physically present but not in the working state, not all steps are required, as the figures show.



Figure 2. Device enumeration and startup sequence for a UMDF driver

At the bottom of Figure 2, the device is not present on the system. When the user plugs in the device, the framework begins by calling the driver’s **IDriverEntry::OnDeviceAdd** callback so that the driver can create a device callback object and a framework DO to represent the device. The framework continues calling the driver’s callback routines by progressing up through the sequence until the device is operational.

Figure 3 shows the callbacks for a KMDF FDO or filter DO that is involved in bringing a device to the fully operational state.



Figure 3. Device enumeration and startup sequence for KMDF FDO or filter DO

At the bottom of Figure 3, the device is not present on the system. When the user inserts the device, KMDF begins by calling the driver’s *EvtDriverDeviceAdd* callback so that the driver can create a DO to represent the device. KMDF continues to call the driver’s callback routines by progressing up through the sequence until the device is operational. Remember that the event callbacks are listed in bottom-up order as shown in Figure 3, so *EvtDeviceFilterRemoveResourceRequirements* is called before *EvtDeviceFilterAddResourceRequirements* and so forth.

Figure 4 shows the callbacks for a bus driver (PDO) that is involved in bringing a device to the fully operational state.



Figure 4. Device addition/startup sequence for PDO

The framework retains the PDO until the corresponding device is physically removed from the system or the bus that enumerated the device is disabled. For example, if a user disables the device in Device Manager but does not physically remove it, KMDF retains the PDO. This requirement is imposed by the underlying WDM model. Thus, the three steps at the bottom of Figure 4 occur only during Plug and Play enumeration—that is, during initial boot, when the user plugs in a new device, and when the bus to which the device is attached enumerates its child devices.

## Device Power-Down and Removal

During device power-down, the sequence of callbacks depends on the role of the DO just as it does during device startup. In general, the power-down and removal sequence involves calling the corresponding “undo” callbacks in the reverse order from which the framework called the methods that it invoked to make the device operational. Drivers perform power-down operations from the top down, so the driver at the top of the device stack performs its power-down tasks first, and the PDO performs its power-down tasks last.

Figure 5 shows the sequence of UMDF callbacks in power-down and removal. The sequence starts at the top of the figure with a device that is in the working power state (D0).



Figure 5. Device power-down and orderly removal sequence for a UMDF driver

Figure 6 shows the sequence of callbacks in power-down and removal for a KMDF FDO or filter DO.



Figure 6. Device power-down and orderly removal sequence for KMDF FDO and filter DO

Figure 7 shows the callbacks in the power-down and removal sequence for a PDO.



Figure 7. Device power-down and orderly removal sequence for PDO

As previously mentioned in “Device Enumeration and Startup” earlier in this paper, the framework does not delete the PDO until the device is physically removed from the system or the bus that enumerated the device is disabled. For example, if a user disables the device in Device Manager or uses the Safely Remove Hardware utility to stop the device but does not physically remove it, KMDF retains the PDO. If the device is later reenabled, KMDF uses the same PDO and begins the startup sequence by calling the *EvtDevicePrepareHardware* callback, as previously shown in Figure 4.

## Surprise Removal

If the user removes a device without warning, by simply unplugging it without using Device Manager or the Safely Remove Hardware utility, the device is considered “surprise removed.” When surprise removal occurs, WDF follows a slightly different removal sequence from the sequence that is used with orderly removal and shutdown. WDF also follows the surprise-removal sequence if any driver in the kernel-mode stack invalidates the device state even if the device is still physically present. A KMDF driver can invalidate the device state by calling **WdfDeviceSetDeviceState**.

Drivers for all removable devices must ensure that the callbacks in both the shutdown and startup paths can handle failure, particularly failures that are caused by hardware removal.

### UMDF Surprise-Removal Sequence

In the surprise-removal sequence, UMDF calls the **IPnpCallback::OnSurpriseRemoval** callback to notify the driver that the device has been unexpectedly removed. This callback is not guaranteed to occur in any particular order with the other callbacks in the removal sequence.

Generally, the driver should avoid accessing the hardware in the remove path. The reflector times out the driver if an attempt to access the hardware waits indefinitely. Figure 8 shows the surprise-removal sequence for a UMDF driver.



Figure 8. Surprise-removal sequence for a UMDF driver

### KMDF Surprise-Removal Sequence

The framework can call the *EvtDeviceSurpriseRemoval* callback at any time before, during, or even after the power-down sequence. For example, if the user unplugs the device during an idle power-down, the framework can call the *EvtDeviceSurpriseRemoval* callback in the middle of the sequence. There is no guarantee on the order in which *EvtDeviceSurpriseRemoval* is called in relation to the other power-down callbacks.

KMDF destroys the DO after the *EvtDeviceSurpriseRemoval* callback has returned and the last handle to the WDF DO has been closed.

Any attempts to access the hardware should not block indefinitely, but should be subject to time-outs or a watchdog timer.

Figure 9 shows the surprise-removal sequence for a KMDF driver.



Figure 9. Surprise-removal sequence for a KMDF driver

# How to Implement Plug and Play and Power Management in WDF Drivers

The rest of this paper provides sample code that shows how to implement Plug and Play and power management in several types of drivers:

* Software-only drivers.
* Simple function drivers, such as UMDF protocol function drivers and KMDF hardware function drivers that do not support idle or wake.
* Hardware function drivers that support idle and wake.

Plug and Play and power management implementation is more complex in each successive example, and each example builds upon the information in the previous examples. You can add the Plug and Play and power code to your own driver in a similar incremental way. Even if your device supports advanced capabilities such as idle or wake, you can start by implementing the simple features. When these features work correctly, you can implement additional callbacks to support the more complex features.

For each type of driver, the discussion covers:

* The type of driver and the Plug and Play and power management features that the driver implements.
* The framework methods that the driver calls and the event callbacks that the driver implements to support the Plug and Play and power management features.
* Sample code that shows the implementation of those features.
* The actions that the framework takes in response to various example Plug and Play and power events for this driver type.

In each sample listing, the significant lines of code are in bold.

# Plug and Play and Power Management in Software-Only Drivers

A software-only driver is a driver that does not control any hardware, either directly or through a protocol such as USB. For example, a root-enumerated function driver is a software-only driver and some filter drivers are software-only drivers. The devnode for a root-enumerated function driver is enumerated from the root of the device tree, and the driver is not associated with any hardware. Software-only drivers can be written for either user mode or kernel mode.

A root-enumerated, software-only KMDF driver creates an FDO and thus is by default considered the PPO for its stack. However, because the driver does not control physical hardware, it does not perform any specific power policy actions—the WDF defaults are sufficient to manage power policy.

Filter drivers are rarely PPOs for their stacks. However, if a filter driver is the power policy manager for its stack, the driver notifies the framework as part of DO initialization so that WDF can initialize the DO appropriately. If the framework’s defaults are otherwise adequate for the driver, the driver does not require any additional Plug and Play or power callbacks. The framework can manage Plug and Play and power for the driver, just as for the software-only function driver. If the framework’s defaults are not adequate, a filter driver can implement Plug and Play and power callbacks to satisfy its requirements.

By default, the framework implements power management for all I/O queue objects that are children of FDOs and PDOs. Queues that are associated with filter DOs are not power managed. Because device hardware is not accessible when the device is in a state other than D0, the framework dispatches requests from a power-managed queue to the driver only when the device is in the D0 state.

Software-only drivers, by definition, do not access any device hardware. Therefore, such drivers should typically disable power management for all their queues. Disabling power management for the queues means that the framework dispatches requests to the driver regardless of the state of the underlying device hardware. The driver can then process the request as usual and forward it, if necessary, to the next lower driver. A driver disables power management for a queue object when it creates the queue.

## UMDF Example: Plug and Play in a Software-Only Filter Driver

The USB Filter sample requires no special code to handle plug and play or power management. Instead, the driver simply:

* Initializes the DO as a filter.
* Indicates that the DO does not own power policy. This call is not required because UMDF by default assumes that the driver is not the PPO.
* Creates non-power-managed queues.

All these tasks are part of **IDriverEntry::OnDeviceAdd** processing. In the USB Filter sample driver, this processing includes the Initialize method, which is implemented on the device callback object in Device.cpp, as Listing 1 shows.

Listing 1. Sample PnP initialization in a UMDF filter driver

HRESULT CMyDevice::Initialize(

\_\_in IWDFDriver \* FxDriver,

\_\_in IWDFDeviceInitialize \* FxDeviceInit

)

{

IWDFDevice \*fxDevice;

HRESULT hr;

FxDeviceInit->SetLockingConstraint(None);

**FxDeviceInit->SetFilter();**

FxDeviceInit->SetPowerPolicyOwnership(FALSE);

{

IUnknown \*unknown = this->QueryIUnknown();

hr = FxDriver->CreateDevice (FxDeviceInit, unknown, &fxDevice);

unknown->Release();

}

if (SUCCEEDED(hr)) {

m\_FxDevice = fxDevice;

fxDevice->Release();

}

return hr;

}

The Initialize function in Listing 1 initializes and creates the framework’s DO. The significant steps here are in bold. The call to **IWDFDeviceInitialize::SetFilter** tells the framework that the driver acts as a filter, so the framework should change its default for request types that the driver does not handle. Instead of failing such requests, the framework passes them to the next lower driver. The call to **IWDFDeviceInitialize::SetPowerPolicyOwnership** indicates to the framework that the driver does not own power policy for the device. This call is not required in this driver, but is included for demonstration purposes.

The only other required step in a UMDF filter driver is to create non-power-managed queues. The driver does this when it calls **IWDFDevice::CreateIoQueue**, as the following shows:

hr = FxDevice->CreateIoQueue( unknown,

TRUE, // bDefaultQueue

WdfIoQueueDispatchParallel,

FALSE, // bPowerManaged

TRUE, // bAllowZeroLengthRequests

&fxQueue

);

In this call, the important item is the fourth parameter (*bPowerManaged*), which indicates whether the framework should manage power for the queues. A software-only driver passes FALSE for this parameter so that the framework dispatches requests to the driver whether or not the device is in the working power state.

## KMDF Example: Plug and Play in a Software-Only Filter Driver

As described earlier, the framework automatically handles Plug and Play and power management tasks for software-only drivers by default.

Listing 2, which is adapted from the Toaster Filter sample, shows a basic *EvtDriverDeviceAdd* function for a software-only KMDF filter driver. This function sets up two Plug and Play or power management features, which are highlighted in this listing:

* An optional cleanup event callback for the DO.
* A non-power-managed I/O queue.

Listing 2. Sample PnP initialization in a KMDF software-only filter driver

NTSTATUS FilterEvtDriverDeviceAdd(

IN WDFDRIVER Driver,

IN PWDFDEVICE\_INIT DeviceInit)

**{**

NTSTATUS status **=** STATUS\_SUCCESS;

PFDO\_DATA fdoData;

WDF\_IO\_QUEUE\_CONFIG queueConfig;

WDF\_OBJECT\_ATTRIBUTES fdoAttributes;

WDFDEVICE hDevice;

**WdfFdoInitSetFilter(DeviceInit)**;

// Initialize the object attributes for our WDFDEVICE.

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&fdoAttributes, FDO\_DATA);

**fdoAttributes.EvtCleanupCallback = FilterEvtDeviceContextCleanup;**

// Create a framework device object.

status = WdfDeviceCreate(&DeviceInit, &fdoAttributes, &hDevice);

if (!NT\_SUCCESS(status)) {

return status;

}

status = WdfDeviceCreateDeviceInterface(hDevice,

&GUID\_DEVINTERFACE\_FILTER,

NULL);

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

}

// Initialize the default queue.

WDF\_IO\_QUEUE\_CONFIG\_INIT\_DEFAULT\_QUEUE(&queueConfig,

WdfIoQueueDispatchParallel);

**queueConfig.PowerManaged = FALSE;**

// Specify event processing callbacks.

queueConfig**.**EvtIoWrite **=** FilterEvtIoWrite;

// Create the queue

status = WdfIoQueueCreate(hDevice,

&queueConfig,

WDF\_NO\_OBJECT\_ATTRIBUTES,

NULL);

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

}

return STATUS\_SUCCESS;  
}

The *EvtDriverDeviceAdd* function in Listing 2 is called with a pointer to the WDF driver object and a pointer to a WDFDEVICE\_INIT structure. The WDFDEVICE\_INIT structure is used to initialize a variety of characteristics that are applied when the DO is created.

To indicate that the DO represents a filter driver, the driver passes the WDFDEVICE\_INIT pointer to **WdfFdoInitSetFilter**. As a result, the framework changes its default processing for any I/O queues that are children of the DO. Instead of failing request types that the driver does not handle, the framework passes them to the next lower driver. In addition, the framework creates I/O queues that are not power managed.

The driver registers the DO’s context type (FDO\_DATA) as part of the WDF\_OBJECT\_ATTRIBUTES structure. By filling in the **EvtCleanupCallback** member of this same structure, the driver registers to be called at its FilterEvtDeviceContextCleanup function when the DO is deleted. A driver should implement this callback if, for example, it has allocated memory other than what the standard WDF object context structures provide and must free that memory when the DO is deleted.

The driver then creates the DO and the device interface by calling **WdfDeviceCreate** and **WdfDeviceCreateDeviceInterface**, respectively.

Following this, the driver initializes a WDF\_IO\_QUEUE\_CONFIG structure for its default queue, providing a callback for handling write requests. It sets the **PowerManaged** field of the WDF\_IO\_QUEUE\_CONFIG structure to FALSE to indicate that the I/O queue that is being created should not be power managed. The driver passes this structure as input to **WdfIoQueueCreate** to create a single default queue to handle requests for the driver.

Creating a non-power-managed queue means that the framework calls the driver whenever a write request arrives, regardless of the power state of the device.

## Framework Actions for Software-Only Drivers

In software-only and filter drivers, the framework handles nearly all Plug and Play and power management operations. Because the driver does not control any hardware, it is not required to provide any additional event callbacks. The framework automatically processes all power management requests correctly.

The UMDF driver’s only Plug and Play callback function is **IDriverEntry::OnDeviceAdd**, and the KMDF driver’s only Plug and Play callback is *EvtDriverDeviceAdd*. Although the KMDF example also provides an optional *EvtCleanupCallback*, nothing in the earlier example driver code actually requires implementing this callback.

The sample drivers disable power management for their queues. As a result, the framework does not automatically hold and release the queue based on arriving Plug and Play and power management events. Instead, the driver continues to receive I/O requests regardless of the Plug and Play and power state of the device.

# Plug and Play and Power Management in Simple Hardware Drivers

A driver that supports hardware—such as a UMDF protocol function driver or a KMDF hardware function driver—differs from a software-only driver in the following ways:

* A driver that supports hardware must initialize its device to a known state each time that the device enters D0, including during system startup.

This known state is typically fully “reset.” If the device supports interrupts or direct memory access (DMA), interrupts are disabled and DMA is stopped.

* Most drivers that interact directly with their device’s hardware create one or more power-managed I/O queues.

The framework automatically stops dispatching I/O requests from the power-managed queues whenever the device hardware is not accessible, such as when the device is powered down.

A simple hardware driver, as described in this section, manages its device hardware through power-up initialization and power-down teardown and uses power-managed queues.

Most KMDF hardware drivers manage hardware resources and device interrupts from their devices and thus must support callback functions to process resources, to enable and disable interrupts, and to handle interrupts when they occur.

Advanced features, such as device idle and wake, are described in “Advanced Power Management” later in this paper.

## Device Power-Up Initialization and Power-Down Teardown

The framework provides hardware function drivers the opportunity to perform device initialization whenever the device enters the D0 state and to perform teardown whenever the device leaves the D0 state. Each time a device enters D0, the framework calls the driver’s D0 entry callback:

* For a UMDF driver, the framework calls **IPnpCallback::OnD0Entry**.
* For a KMDF driver, the framework calls *EvtDeviceD0Entry*.

For a KMDF driver, *EvtDeviceD0Entry* is called before the driver’s *EvtInterruptEnable* callback. Therefore, interrupts have not yet been enabled for the device and the device is not yet connected to the driver’s *EvtInterruptIsr* callback. During *EvtDeviceD0Entry*, drivers must not enable interrupts on their device or do anything that causes their device to interrupt. This is important to avoid potential “interrupt storms.” The same is true for the *EvtDevicePrepareHardware* callback. If the device requires initialization after its interrupt is connected, the driver should register *EvtDeviceD0EntryPostInterruptsEnabled*.

The framework calls these callbacks after the bus driver has powered up the device, so the device hardware is accessible to the driver. Every device is powered up implicitly whenever the device is first detected, such as during system startup, and after the PnP manager stops the device to rebalance resources. Therefore, the framework always calls the D0 entry callbacks during startup, after calling the prepare-hardware callbacks.

Within the D0 entry callback, a driver performs any required hardware-related tasks each time that the device enters the D0 state. Such tasks might include downloading firmware to the device and initializing the device to a known state or restoring the state that was previously saved during power-down.

Each time a device is about to leave D0, the framework calls the device’s driver at its D0 exit callback:

* For a UMDF driver, the framework calls **IPnpCallback::OnD0Exit**.
* For a KMDF driver, the framework calls *EvtDeviceD0Exit*.

For a KMDF driver, the framework calls *EvtDeviceD0Exit* after it calls *EvtInterruptDisable*, so device interrupts have been disabled and disconnected from the driver’s *EvtInterruptIsr* callback. As with D0 entry, if the device requires teardown before its interrupts are disabled, the driver should register *EvtDeviceD0ExitPreInterruptsDisabled*.

DuringD0 exit processing, a driver performs tasks that are related to power-down, such as saving internal device state. The device hardware is still accessible in these callbacks because the device is still in the D0 power state.

## Power Management for Queues in Hardware Function Drivers

By default, the framework manages power for the I/O queue objects that are children of FDOs and PDOs. As previously described, when the framework manages power for a queue, it dispatches requests from the queue to the driver’s I/O callback functions only when the device hardware is accessible and in the D0 state. Letting the framework manage power for a queue means that the driver is not required to maintain device state or to check device state each time it receives an I/O request from the queue. Instead, it can process the request and access device hardware as required until the request has been completed.

Of course, not all I/O requests that a driver receives require access to device hardware. For example, a driver can often handle some device I/O control requests without accessing the hardware. Such a driver should create two queues—one power-managed queue and one non-power-managed queue. The driver configures the non-power-managed queue to receive all the device I/O control requests from the framework. The framework dispatches requests from this queue regardless of the power state of the device. The driver inspects each request, handles the request if possible, and—if the device is not in the working state—forwards any requests that it cannot handle to the power-managed queue.

Drivers typically create and configure their I/O queues during add-device processing—that is, in a UMDF driver’s **IDriverEntry::OnDeviceAdd** callback or a KMDF driver’s *EvtDriverDeviceAdd* callback.

To disable power management for a queue:

* A UMDF driver sets the *bPowerManaged* parameter of the **IWDFDevice::CreateIoQueue** method to FALSE when it creates the queue.
* A KMDF driver sets the **PowerManaged** field of the WDF\_IO\_QUEUE\_CONFIG structure to **WdfFalse** when it creates the queue.

Drivers that handle some I/O requests that require hardware access and other requests that do not require hardware access should create multiple queues and sort their requests on this basis. The UMDF Fx2\_Driver and KMDF Osrusbfx2 samples create both power-managed and non-power-managed queues.

## UMDF Example: Plug and Play and Power Code in a Protocol Function Driver

A UMDF function driver that manages device hardware is different from a software-only driver in several ways:

* The driver typically creates one or more power-managed queues.
* The driver implements the **IPnpCallback** and **IPnpCallbackHardware** interfaces as required on the device callback object to perform tasks that are related to the Plug and Play and power state of the device.
* The driver must determine whether it should be the PPO for the device stack. By default, UMDF drivers do not own power policy.

Tables 3 and 4 summarize the methods in the **IPnpCallback** and **IPnpCallbackHardware** interfaces.

Table 3. IPnpCallbackHardware Methods

| Name | Description | When called |
| --- | --- | --- |
| **OnPrepareHardware** | Prepares the device and driver to enter the working state after enumeration or resource rebalance. | After **IDriverEntry::OnDeviceAdd** returns and before the device enters the working power state. |
| **OnReleaseHardware** | Prepares the device and driver before system shutdown or resource rebalance. | After the device exits from the working power state but before its queues are purged. |

The methods in **IPnpCallbackHardware** provide for driver actions when its device is added to or removed from the system and when system resources are rebalanced.

Table 4. IPnpCallback Methods

| Name | Description | When called |
| --- | --- | --- |
| **OnD0Entry** | Performs required tasks for device to begin operation. | Immediately after the device enters the working power state. |
| **OnD0Exit** | Performs required tasks for device to end operation. | Immediately before the device exits the working power state. |
| **OnSurpriseRemoval** | Cleans up after device is unexpectedly removed. | Immediately after the device is removed unexpectedly. |
| **OnQueryRemove** | Provides the opportunity for the driver to veto a request to remove the device. | While the device is in the working state, before the device is physically removed. |
| **OnQueryStop** | Provides the opportunity for the driver to veto a request to stop the device to rebalance resources. | While the device is in the working state, before it is stopped to rebalance resources. |

The **IPnpCallback** interface includes methods that are required to support the most common Plug and Play and power events, such as doing any initialization that is required after the device is powered on and the corresponding teardown that is required before the device powers down.

The Fx2\_Driver sample is a UMDF protocol function driver that implements both **IPnpCallback** and **IPnpCallbackHardware** on the device callback object. The driver creates a default power-managed queue for read and write requests and a separate power-managed queue that receives only device I/O control requests.

### Power-Managed Queue for a UMDF Driver

To create a power-managed queue, a UMDF driver calls passes TRUE for the *bPowerManaged* parameter to **IWDFDevice::CreateIoQueue**, as follows:

hr = FxDevice->CreateIoQueue( unknown,

TRUE, // bDefaultQueue

WdfIoQueueDispatchParallel,

TRUE, // bPowerManaged

TRUE, // bAllowZeroLengthRequests

&fxQueue

);

The driver typically creates I/O queues in its **IDriverEntry::OnDeviceAdd** callback.

### IPnpCallbackHardware Methods

The framework calls the methods in the **IPnpCallbackHardware** interface on the DO before the device enters D0 and after the device leaves D0.

In the **OnPrepareHardware** method, the driver prepares to communicate with device hardware. It opens a handle to the device and calls internal functions to get information about the USB interfaces and endpoints.

Listing 3 shows the **OnPrepareHardware** method that the Fx2\_Driver sample implements on the device callback object in the Device.cpp source file. To conserve space, error-handling statements have been omitted.

Listing 3. Sample IPnpCallbackHardware::OnPrepareHardware method

HRESULT CMyDevice::OnPrepareHardware(

**\_\_in IWDFDevice \* /\* FxDevice \*/**

)

{

HRESULT hr;

. . . //Code omitted

// Create USB I/O targets and configure them.

**hr = CreateUsbIoTargets();**

if (SUCCEEDED(hr)) {

ULONG length = sizeof(m\_Speed);

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

**&length,**

**&m\_Speed);**

if (FAILED(hr)) {

// Generate trace message.

}

}

. . . //Code omitted

hr = ConfigureUsbPipes();

//

// Clear the seven segment display to indicate that

// hardware preparation is complete.

//

if (SUCCEEDED(hr)) {

hr = IndicateDeviceReady();

}

// Configure continuous reader.

if (SUCCEEDED(hr)) {

hr = ConfigContReaderForInterruptEndPoint();

}

if (SUCCEEDED(hr)) {

. . . //Code omitted

}

return hr;

}

The **OnPrepareHardware** method performs tasks that are required to ready the device for I/O before it enters the working state. These tasks include creating and configuring the USB I/O targets for the driver. The **OnPrepareHardware** method uses the framework’s **IWDFUsbTargetDevice** interface to get information about the USB hardware.

Listing 4 shows the Fx2\_Driver sample’s **OnReleaseHardware** method.

Listing 4. Sample IPnpCallbackHardware::OnReleaseHardware method

HRESULT CMyDevice::OnReleaseHardware(

\_\_in IWDFDevice \* /\* FxDevice \*/

)

{

if (m\_pIUsbTargetDevice) {

m\_pIUsbTargetDevice->DeleteWdfObject();

}

return S\_OK;

}

The **OnReleaseHardware** method performs cleanup tasks that are required when the device leaves the working state. The Fx2\_Driver sample deletes the USB target DO, which in turn deletes all child interface and pipe objects. The hardware does not require any additional service before power-down. For example, the driver does not save any hardware context information.

### IPnpCallback Methods

When the OSR USB Fx2 device powers up, the Fx2\_Driver sample starts its USB I/O target pipes. When the device powers down, the driver stops the target pipes. To perform these tasks, the driver implements the **OnD0Entry** and **OnD0Exit** methods of the **IPnpCallback** interface. In this driver, the other methods of the **IPnpCallback** interface are token implementations.

In the sample, the only tasks that these methods perform involve its I/O targets, which the driver stops and restarts when the device enters or leaves the working state. The driver does not actually manipulate device hardware in any of these functions. Instead, it prepares to begin and end handling I/O requests.

Listings 5 and 6 show the code from the Device.cpp source file that implements the **OnD0Entry** and **OnD0Exit** methods.

Listing 5. Sample IPnpCallback::OnD0Entry method

HRESULT STDMETHODCALLTYPE

CMyDevice::OnD0Entry(

\_\_in IWDFDevice \* /\* FxDevice \*/,

\_\_in WDF\_POWER\_DEVICE\_STATE /\* PreviousState \*/

)

{

m\_pIoTargetInterruptPipeStateMgmt->Start();

return S\_OK;

}

In the **OnD0Entry** method, the driver calls **IWDFIoTargetStateManagment::Start** on the continuous reader I/O target. The driver created this target in **IPnpCallbackHardware::OnPrepareHardware**.

The **OnD0Exit** method stops the I/O target, as Listing 6 shows. If a USB input pipe is configured with a continuous reader, the **OnD0Exit** method must stop the target. If a continuous reader is not configured on the pipe, the driver is not required to stop the target, although it might choose to stop the target for other reasons.

Listing 6. Sample IPnpCallback::OnD0Exit method

HRESULT STDMETHODCALLTYPE

CMyDevice::OnD0Exit(

**\_\_in IWDFDevice /\* FxDevice \*/,**

**\_\_in WDF\_POWER\_DEVICE\_STATE /\* NewState \*/**

)

{

m\_pIoTargetInterruptPipeStateMgmt->Stop(  
 WdfIoTargetCancelSentIo);

return S\_OK;

}

## KMDF Example: Plug and Play and Power Code in a Simple Hardware Function Driver

A simple KMDF hardware function driver that manages a device through startup and shutdown requires only a few more callback functions than a software-only driver requires.

The code in this section is adapted from the Osrusbfx2 sample. It includes support for the following features in addition to those required for a software-only driver:

* EvtDevicePrepareHardware callback.
* EvtDeviceD0Entry and EvtDeviceD0Exit callbacks.
* Four I/O queues, three of which are power managed.

By providing these few functions, this driver fully supports Plug and Play and power management for its device.

### KMDF Example: Register Callbacks and Set Up Power-Managed Queues

Listing 7 shows how the Osrusbfx2 driver registers the fundamental Plug and Play and power management callbacks and creates power-managed I/O queues. This function appears in the Device.c source file.

Listing 7. EvtDriverDeviceAdd for simple hardware function driver

NTSTATUS OsrFxEvtDeviceAdd(

IN WDFDRIVER Driver,

IN PWDFDEVICE\_INIT DeviceInit

)

{

WDF\_PNPPOWER\_EVENT\_CALLBACKS pnpPowerCallbacks;

WDF\_OBJECT\_ATTRIBUTES attributes;

NTSTATUS status;

WDFDEVICE device;

WDF\_DEVICE\_PNP\_CAPABILITIES pnpCaps;

WDF\_IO\_QUEUE\_CONFIG ioQueueConfig;

PDEVICE\_CONTEXT pDevContext;

WDFQUEUE queue;

UNREFERENCED\_PARAMETER(Driver);

**WDF\_PNPPOWER\_EVENT\_CALLBACKS\_INIT(&pnpPowerCallbacks);**

**pnpPowerCallbacks.EvtDevicePrepareHardware =**

**OsrFxEvtDevicePrepareHardware;**

**pnpPowerCallbacks.EvtDeviceD0Entry = OsrFxEvtDeviceD0Entry;**

**pnpPowerCallbacks.EvtDeviceD0Exit = OsrFxEvtDeviceD0Exit;**

**WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit,**

**&pnpPowerCallbacks);**

WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoBuffered);

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&attributes,

DEVICE\_CONTEXT);

// Create a framework device object.

status = WdfDeviceCreate(&DeviceInit, &attributes, &device);

if (!NT\_SUCCESS(status)) {

return status;

}

pDevContext = GetDeviceContext(device);

// Set SurpriseRemovalOK in the Device Capabilities so

// that a user-mode popup does not appear on Windows 2000 when

// the user surprise-removes the device.

**WDF\_DEVICE\_PNP\_CAPABILITIES\_INIT(&pnpCaps);**

**pnpCaps.SurpriseRemovalOK = WdfTrue;**

**WdfDeviceSetPnpCapabilities(device, &pnpCaps);**

// Create a default queue.

. . . //Code omitted

// Create a separate sequential queue for read requests.

WDF\_IO\_QUEUE\_CONFIG\_INIT( &ioQueueConfig,

WdfIoQueueDispatchSequential);

ioQueueConfig.EvtIoRead = OsrFxEvtIoRead;

**ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;**

status = WdfIoQueueCreate( device,

&ioQueueConfig,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&queue // queue handle

);

if (!NT\_SUCCESS (status)) {

return status;

}

status = WdfDeviceConfigureRequestDispatching(

device,

queue,

WdfRequestTypeRead

);

if(!NT\_SUCCESS (status)){

return status;

}

// Create another sequential queue for write requests.

. . . //Code omitted

// Create a manual I/O queue. We retrieve requests from this

// queue only when the device sends an interrupt.

WDF\_IO\_QUEUE\_CONFIG\_INIT(&ioQueueConfig, WdfIoQueueDispatchManual);

**ioQueueConfig.PowerManaged = WdfFalse;**

status = WdfIoQueueCreate(device,

&ioQueueConfig,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&pDevContext->InterrputMsgQueue

);

if (!NT\_SUCCESS(status)) {

. . . //Additional code omitted

}

return status;

}

In the example, the first highlighted lines are related to the Plug and Play and power management callbacks. The driver initializes a WDF\_PNPPOWER\_EVENT\_CALLBACKS structure and fills in pointers to its *EvtDevicePrepareHardware*, *EvtDeviceD0Entry*, and *EvtDeviceD0Exit* callback functions. This driver manages a USB device, so it implements only an *EvtDevicePrepareHardware* callback without a corresponding *EvtDeviceReleaseHardware* callback. In a USB driver, the *EvtDevicePrepareHardware* callback selects interfaces and retrieves other information about the USB device before the device enters the working power state. However, no corresponding teardown is required, so the driver does not implement *EvtDeviceReleaseHardware*. Drivers for device types other than USB typically implement both of these callbacks.

The driver then sets the WDF\_PNPPOWER\_EVENT\_CALLBACKS structure into the WDFDEVICE\_INIT structure by calling **WdfDeviceInitSetPnpPowerEventCallbacks**. The WDFDEVICE\_INIT structure—and thus the callbacks that were just described—is associated with the DO when the driver calls **WdfDeviceCreate**.

The second group of highlighted lines sets the device’s Plug and Play capabilities. After the driver creates the DO, it initializes the WDF\_DEVICE\_PNP\_CAPABILITIES structure by setting the **SurpriseRemovalOK** field to **WdfTrue** and then calls **WdfDeviceSetPnpCapabilities** to pass this information to the framework. This setting indicates that users can safely remove the device without using the Safely Remove Hardware utility.

The sample driver creates four I/O queues, three of which are power managed. The **PowerManaged** field in the WDF\_IO\_QUEUE\_CONFIG structure indicates whether a queue is power managed. If the driver does not set this field, KMDF uses the default value and therefore creates power-managed queues based on the DO’s role as the FDO for the device stack. To create a queue that is not power managed, the driver must explicitly set this field to **WdfFalse**.

The next highlighted line of code registers an *EvtIoStop* callback for one of the power-managed queues by setting the **EvtIoStop** field of the WDF\_IO\_QUEUE\_CONFIG structure. The framework invokes *EvtIoStop* for a power-managed queue before the device leaves the working state. This function handles any pending I/O requests as appropriate for the device and the driver.

The final highlighted line of code shows how the driver sets the **PowerManaged** field in the WDF\_IO\_QUEUE\_CONFIG structure for its non-power-managed queue.

### KMDF Example: D0 Entry and D0 Exit Callbacks

The framework calls the driver’s *EvtDeviceD0Entry* callback immediately after the driver enters the D0 state and calls the *EvtDeviceD0Exit* callback immediately before the driver exits the D0 state.

*EvtDeviceD0Entry* must perform any operations that are required before the device can be used. The framework calls this callback each time that the hardware must be initialized or reinitialized. Listing 8 shows the *EvtDeviceD0Entry* function from the Osrusbfx2 sample driver.

Listing 8. EvtDeviceD0Entry callback for simple hardware function driver

NTSTATUS OsrFxEvtDeviceD0Entry(

IN WDFDEVICE Device,

IN WDF\_POWER\_DEVICE\_STATE PreviousState

)

{

PDEVICE\_CONTEXT pDeviceContext;

NTSTATUS status;

PAGED\_CODE();

pDeviceContext = GetDeviceContext(Device);

status = WdfIoTargetStart(

WdfUsbTargetPipeGetIoTarget( pDeviceContext->InterruptPipe)

);

return status;

}

The Osrusbfx2 driver’s *EvtDeviceD0Entry* callback simply starts the driver’s I/O targets, as Listing 8 shows.

The *EvtDeviceD0Exit* callback performs any operations that are required before the device leaves the D0 state, such as saving hardware state. The device is still in D0 when *EvtDeviceD0Exit* runs, so the driver can touch the hardware. The Osrusbfx2 driver’s *EvtDeviceD0Exit* callback is shown in Listing 9.

Listing 9. EvtDeviceD0Exit callback for simple hardware function driver

NTSTATUS OsrFxEvtDeviceD0Exit(

IN WDFDEVICE Device,

IN WDF\_POWER\_DEVICE\_STATE TargetState

)

{

PDEVICE\_CONTEXT pDeviceContext;

PAGED\_CODE();

pDeviceContext = GetDeviceContext(Device);

WdfIoTargetStop(WdfUsbTargetPipeGetIoTarget(

pDeviceContext->InterruptPipe),

WdfIoTargetCancelSentIo

);

return STATUS\_SUCCESS;

}

The sample driver’s *EvtDeviceD0Exit* callback simply undoes the actions of *EvtDeviceD0Entry*. Therefore, it stops the I/O targets and returns STATUS\_SUCCESS.

## Framework Actions for a Simple Hardware Function Driver

As in the software-only driver examples, the frameworks implement almost all Plug and Play and power management support for the sample drivers that were just described. Because these are function drivers, the framework automatically creates and manages Plug and Play, power management, and power policy state machines. The driver requires code only to manage the device hardware.

The framework calls the driver’s prepare-hardware callback after the PnP manager discovers a device supported by the driver and after the driver’s add-device callback returns:

* For both UMDF and KMDF drivers, this function performs any initialization that is required before the device enters the D0 state.

The prepare-hardware callbacks of USB drivers, such as the Fx2\_Driver and Osrusbfx2 samples, should call methods that return information about the device and should also configure the interfaces on the device.

* For a KMDF driver, the parameters to the prepare-hardware callback include a handle to the hardware resources that have been assigned to the driver.

A driver that manages device resources can use KMDF helper functions to access and manipulate the resource lists.

Just before the device enters D0, the framework calls the driver’s D0 entry callback. One of the parameters to this function for both UMDF and KMDF drivers is the previous power state from which the device is transitioning. Drivers typically ignore this value and initialize the device in the same way regardless of the previous power state. The value for this parameter is one of the following enumeration constants of the WDF\_POWER\_DEVICE\_STATE type:

**WdfPowerDeviceUnspecified   
WdfPowerDeviceD0  
WdfPowerDeviceD1  
WdfPowerDeviceD2  
WdfPowerDeviceD3  
WdfPowerDeviceD3Final  
WdfPowerDevicePrepareForHibernation**

If the device is powering up for the first time, the framework passes **WdfPowerDeviceUnspecified**.

The framework also passes a device power state to thedriver’s D0 exit callback. In this case, the device state indicates the power state to which the device is transitioning upon exit from D0. Two of the possible target states might be unfamiliar to you:

* WdfPowerDeviceD3Final
* WdfPowerDevicePrepareForHibernation (KMDF only)

The framework passes **WdfPowerDeviceD3Final** as the target device power state to indicate that this is a transition to D3 as part of system shutdown or device removal. In this case, the driver must do whatever unique activities are necessary to prepare for shutdown, such as saving state to a disk or other nonvolatile medium.

Before a device leaves the D0 state, the framework stops any power-managed I/O queues that are associated with that device. After the device reenters the D0 state, the framework resumes the power-managed I/O queues.

KMDF drivers for certain storage devices might receive **WdfPowerDevicePrepareForHibernation** as the target state if the device is in the hibernation path and the system is preparing to hibernate. The hibernation path includes the device to which the system writes the hibernation file and any other devices along the path from the root to that device, which are required to maintain power to that device.

The framework passes **WdfPowerDevicePrepareForHibernation** only if the driver has both called **WdfDeviceSetSpecialFileSupport** and received notification that it is in the hibernation path—and then only if the target power state for the system is S4.

When a KMDF driver’s *EvtDeviceD0Exit* callback is called with the **WdfPowerDevicePrepareForHibernation** target state, the driver should prepare the device for hibernation by doing everything necessary to put the device into D3 except powering it off. This includes saving any state that the driver requires to return the device to the D0 state after the system resumes from hibernation. The driver must not power off the device. The system uses the device when it saves the hibernation file to disk immediately before entering the S4 state.

# Advanced Power Management

As the examples in the previous section show, the framework handles most of the work in implementing Plug and Play and power management, even for a driver that supports a hardware device. This section describes how to go beyond the basics to add support for two advanced features:

* Device idle support

A driver can power down its device when the device becomes idle and the system remains in the working state (S0).

* Device wake support

Some devices can bring themselves, and perhaps the system, into a fully powered working state from a lower-powered state. Properly supporting wake requires specific capabilities in both the device hardware and the driver.

WDF v1.9 and later versions support idle and wake functionality for both KMDF and UMDF drivers. For a UMDF example that implements idle and wake, see “Selective Suspend in USB Drivers” on the WHDC Web site.

## Device Power-Down Idle Support

In many circumstances, powering down a device when it is idle—but while the system remains in the S0 state—has significant advantages:

* Idle support saves power.
* Idle support 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. Adding device idle support requires only a few extra callbacks beyond those that are required for basic Plug and Play support.

### Idle Settings and Management

To configure idle support, a driver sets the idle characteristics for the device and registers them with the framework.

* A UMDF driver specifies idle settings by calling the **IWDFDevice2::AssignS0IdleSettings**.
* A KMDF driver specifies idle settings in the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure and calls **WdfDeviceAssignS0IdleSettings** to register the setting with the framework.

Table 5 lists the idle characteristics that a driver can specify.

Table 5. Device Idle Settings

|  |  |  |
| --- | --- | --- |
| Setting | Description | Possible values |
| **Enabled** | Whether to enable device power-down on idle. | **WdfTrue WdfFalse WdfDefault** (enabled unless explicitly disabled by a user with administrator privileges) |
| **IdleCaps** | Whether the driver supports idle and whether the device and driver support wake in S0.  For a USB driver, whether the device supports USB selective suspend. USB drivers must not specify **IdleCanWakeFromS0**. | **IdleCannotWakeFromS0 IdleCanWakeFromS0 IdleUsbSelectiveSuspend** |
| **DxState** | The device power state to which the framework transitions the idle device. | **PowerDeviceD0 PowerDeviceD1 PowerDeviceD2 PowerDeviceD3** (default if **IdleCaps** is set to **IdleCannotWakeFromS0**) |
| **IdleTimeout** | The amount of time, in milliseconds, that must elapse without receiving an I/O request before the framework considers the device idle. | ULONG or **IdleTimeoutDefaultValue** (currently set to 5000 milliseconds or 5 seconds) |
| **UserControlOfIdleSettings** | Whether the framework provides a property page in Device Manager to allow administrators to control the idle policy for the device. | **IdleDoNotAllowUserControl** **IdleAllowUserControl** |
| **PowerUpIdleDeviceOnSystemWake (KMDF 1.9 and later only)** | Whether the device returns to D0 when the system returns to S0. Valid only if **IdleCaps** is set to **IdleCannotWakeFromS0**. | **WdfTrue** **WdfFalse** **WdfDefault** (same as **WdfFalse**) |

A driver that supports idle but does not support wake from S0 should specify **IdleCannotWakeFromS0** as the **IdleCaps** value. In addition, a KMDF driver can set **PowerUpIdleDeviceOnSystemWake** to **WdfTrue** to indicate that the framework should return the device to the working state when the system returns to S0. By default, the framework returns devices to the working state only when software accesses the device.

A driver for a USB device that supports selective suspend should specify **IdleUsbSelectiveSuspend**.

If the driver specifies **IdleUsbSelectiveSuspend** or **IdleCanWakeFromS0**, the framework uses the reported power capabilities for the device as the default **DxState**.

If the driver specifies **IdleCannotWakeFromS0**, the framework sets **PowerDeviceD3** as the default **DxState**.

A driver can register S0 idle settings any time after it creates the DO. Although most drivers initially enable idle support and set idle characteristics during add-device processing, this may not always be possible or even desirable. If a driver supports multiple devices or device versions, the driver might not know whether the device is capable of wake from S0 until it interrogates its hardware. Such drivers can wait to register idle settings until the prepare-hardware callback.

A driver can change the idle time-out value, the device state in which the device idles, and whether device idle support is enabled any time after its initial call to **IWDFDevice2::AssignS0IdleSettings** (in a UMDF driver) or **WdfDeviceAssignS0IdleSettings** (in a KMDF driver) by simply calling the method again with new values. The same is also true for wake from S0 support for most device types. However, a USB driver must indicate whether the device supports selective suspend the first time that it sets idle characteristics and cannot change this value thereafter. The framework does not recognize changes in selective suspend support in subsequent calls to these methods.

**Note** Whether an individual device can support wake from S0 depends on the capabilities of both the device and the slot or system to which the device is attached. Therefore, a call to assign idle settings that specifies **IdleCanWakeFromS0** can fail with STATUS\_POWER\_STATE\_INVALID in KMDF or HRESULT\_FROM\_NT (STATUS\_POWER\_STATE\_FROM\_NT) in UMDF on hardware configurations where wake from S0 is not supported. You must ensure that this error does not result in a failure to load your driver. If one of the initialization callbacks—such as the add-device or prepare-hardware callback—returns this value to the framework, the framework disables the device.

Sometimes, a device should not be powered down even if no I/O requests are present within the time-out period. A driver can prevent the framework from powering down an idle device in such situations and can later resume idle power-down. A UMDF driver calls **IWDFDevice2::StopIdle** and **IWDFDevice2::ResumeIdle**, and a KMDF driver calls **WdfDeviceStopIdle** and **WdfDeviceResumeIdle** to prevent and re-enable idle power-down.

For example, the KMDF Serial sample does not power down its idle device if a handle is open. The Serial sample calls **WdfDeviceStopIdle** when it receives an open request and calls **WdfDeviceResumeIdle** when it receives a close request. The same is true for USB drivers. This behavior, however, is not appropriate for many other drivers, because most drivers always have an open handle.

The stop and resume idle methods manage a reference count on the device, so if your driver calls stop-idle several times, the device will not go idle until the driver has called resume-idlethe same number of times.

If the device is already in its low-power idle state, the stop-idle method causes the framework to return the device to the D0 state. If the device is in the D0 state, the resume-idle method does not cause the framework to restart the device; instead, resume-idle restarts the idle time-out timer.

The stop-idle method does not prevent the framework from transitioning the device to a sleep state when the system changes to an S*x* sleep state. Its only effect is to prevent transitions to D*x* sleep states while the system is in the S0 working state.

The framework integrates its idle power-down handling with the driver’s other Plug and Play and power management activities. The framework transitions the device to the power state that the driver most recently specified in the assign idle settings method when all of the following conditions are met:

* Idle support is enabled.
* The driver has not deactivated idling by calling the stop-idle method.
* The time-out period expires.
* No I/O requests are active on the device.

The framework returns the device to D0 whenever one of the following occurs:

* A new I/O request arrives at any of the device’s power-managed queues.
* The driver calls the stop-idle method.
* The driver disables idle support by calling the assign S0 idle settings method and setting *Enabled* to **WdfFalse**.
* The system transitions to a system power state that is incompatible with the device power state.

### How to Choose Idle Times and Idle States

A few words on how to choose appropriate idle times and device power states are appropriate here. In general the latency differences between returning a device to D0 from D1, D2, and D3 are on the order of a few milliseconds. There are a few exceptions. Video display devices, for example, can demonstrate differences in latency of several seconds. However, for nearly all other devices the difference in latency between a device in D1 and a device in D3 is so short that an end user is unlikely to perceive the difference. Consequently, considering the greater power savings that is achieved by idling a device in its power-off state, the framework by default transitions idle devices to D3.

Formerly, some vendors hesitated to implement idle support for their devices because they believed that users perceived any latency in their devices as decreased performance. However, this approach prevents these devices from achieving the power, heat, and noise savings that idle support can provide. Microsoft studies have shown that users perceive almost any latency as acceptable if it occurs only when the machine—or perhaps the device—is completely idle. To prevent the device from prematurely entering the idle state, increase the idle time-out value by calling **WdfDeviceAssignS0IdleSettings** in a KMDF driver or **IWDFDevice2::AssignS0IdleSettings** in a UMDFdriver each time that the device becomes busy.

## Device Wake Support

The system power states in which a device can trigger a wake signal depend on the design of the device and the design of the system. Three different models for using the wake signal on a device are in common use:

* Wake from S0: Triggering the wake signal from S0

If the system is in S0 and the device is idle and in a sleep state, an external stimulus causes the device to trigger a wake signal, which in turn causes the framework and the driver to put the device back in the working state. The stimulus could be the click of a mouse button or the insertion of an Ethernet cable for a NIC.

* Wake from S*x*: Triggering the wake signal from S1, S2, S3, or S4

If the system is in a sleep state, the driver can trigger a wake signal to return the system to the working state.

As with wake from S0, the stimulus to trigger the wake signal arrives externally, but the conditions that cause a wake signal are often different. For example, you probably would not want your machine to wake if you insert a network cable, but you might want it to wake if a special packet arrives over the network. For this reason, the framework supports different callbacks for arming a device to wake from S0 and S*x*.

* Wake from S5: Triggering the wake signal from S5

Some devices can trigger the wake signal from S5, which causes the machine to power on from the “entirely off” state.

This capability is often used for remote management over the network. However, this feature is outside the scope of drivers and Windows because it requires BIOS integration. Waking the system from S5 is not considered wake from a software perspective because the machine is not asleep—it is off. As a hardware developer, you must implement wake from S5 in the context of the BIOS, not in the operating system and drivers.

Wake from S*x* is, perhaps, the most common wake category. Consider the following example. While the system is powered down to S3 and the devices that are connected to the system are similarly powered down, a “magic packet” arrives via Ethernet. As a result of receiving this packet, the network adapter hardware—which was appropriately configured before entering its low-power state—triggers a wake event (that is, PME# on the PCI bus) that causes the system to transition to S0. As a result, the system and its connected devices wake and return to the fully operational working state.

Unlike wake from S*x*, wake from S0 is tied to device support for power-down idle. When the device becomes idle, it enters a low-power state while the system remains in S0. Wake from S0 simply means that the device can trigger a wake signal from its low-powered idle state. Wake support and idle support are related in the following ways:

* Devices that support idle power-down while the system is in S0 do not necessarily support wake from S0.
* Devices that support wake from S0 also implicitly support power-down idle.
* Devices that support wake from S*x* do not necessarily support power-down idle, but they might.

The wake setting that your driver should support depends on the scenarios you choose to support for the device. For example, a mouse, a network adapter, and a serial port might support wake as follows:

* A mouse triggers a wake signal when a user moves it or clicks a button. This can occur in S0 or in S*x*, depending on system power policy.
* A network adapter goes idle when a cable is not present and triggers wake in S0 when the user plugs in a cable. A NIC can also trigger wake in S*x* when a “magic packet” arrives. On a system that has a specially designed BIOS, a network adapter could trigger wake from S5 when a custom management application starts the system remotely for servicing.
* The driver has no way to know what type of device is plugged into a serial port. Therefore, whenever the system is in S0 and a handle is open to the serial port, the serial port must be in D0. A serial port can trigger wake from S*x* if the “Ring Indicate” pin is triggered, indicating that a modem is plugged into the port and the associated phone is ringing.

The following sections describe how to implement both wake from S*x* and wake from S0 in a WDF driver.

### How to Implement Wake from Sx

To enable support for wake from S*x*, a driver implements the S*x* wake callback methods and assigns S*x* wake settings.

* A UMDF driver implements the **IPowerPolicyCallbackWakeFromSx** interface on the device callback object to handle wake-up and calls **IWDFDevice2::AssignSxWakeSettings** to assign wake settings.
* A KMDF driver implements the *EvtDeviceArmWakeFromSx*, *EvtDeviceDisarmWakeFromSx*, and *EvtDeviceWakeFromSxTriggered* callbacks, registers them with the framework at device-object creation, and calls **WdfDeviceAssignSxWakeSettings** to assign wake settings.

#### Power Policy Event Callbacks for Wake from S*x*

Table 6 lists the power policy event callbacks for wake from S*x.*

Table 6. Power Policy Event Callbacks for Wake from S*x*

|  |  |  |
| --- | --- | --- |
| Event | UMDF callback | KMDF callback |
| Enable device’s ability to wake system from Sx state | **IPowerPolicyCallbackWakeFromSx:: OnArmWakeFromSx** | *EvtDeviceArmWakeFromSx* |
| Disable device’s ability to wake system from Sx state | **IPowerPolicyCallbackWakeFromSx:: OnDisarmWakeFromSx** | *EvtDeviceDisarmWakeFromSx* |
| Respond to device wake signal in Sx state | **IPowerPolicyCallbackWakeFromSx:: OnWakeFromSxTriggered** | *EvtDeviceWakeFromSxTriggered* |

A UMDF driver requests a callback for one or more of these events by implementing the **IPowerPolicyCallbackWakeFromSx** interface on the device callback object. If the driver supports wake from S*x* but does not require a callback for any power policy events, it is not required to implement the interface.

A KMDF driver registers these callback functions through the WDF\_POWER\_POLICY\_EVENT\_CALLBACKS structure in its *EvtDriverDeviceAdd* callback. This structure is input to the WDFDEVICE\_INIT structure and so must be set up before the driver creates the DO. To initialize the structure, the KMDF driver uses the WDF\_POWER\_POLICY\_EVENT\_CALLBACKS\_INIT macro, which supplies pointers to its *EvtDeviceArmWakeFromSx*, *EvtDeviceDisarmWakeFromSx*, and *EvtDeviceWakeFromSxTriggered* callbacks in the fields of the same names. After the driver fills in the fields that apply to its implementation, it calls **WdfDeviceInitSetPowerPolicyEventCallbacks** to register the callbacks in the WDFDEVICE\_INIT structure.

The framework calls the driver’s S*x* arm-wake callback to request that the driver enable—or arm—its device to wake from S*x*. Within this function, the driver performs the device-specific processing to complete this task. If the driver is not required to perform any device-specific tasks—such as reconfiguring an internal interrupt signal on the device—to arm its device for wake, a UMDF driver can supply a token implementation and a KMDF driver can omit the callback function.

The S*x* disarm-wake callback should reverse any actions that the driver performed in the arm-wake callback. The framework calls it to request that the driver disable—or disarm—its device to wake from S*x*. As with Sx arm-wake, if no device-specific processing is required to disarm the device, a UMDF driver can supply a token implementation and a KMDF driver can omit the callback.

When the driver’s device triggers a wake signal, the framework calls the wake-from-S*x*-triggered callback. Because the framework handles all aspects of waking the system, this callback is merely informative. Consequently most UMDF drivers provide only a token implementation and most KMDF drivers do not register for this callback.

**Note** The framework calls the Sx wake-triggered and S0 wake-triggered callbacksonly if the system’s BIOS and the motherboard are implemented correctly and work perfectly—which is sometimes not the case. If correct operation of the driver depends on knowing when the device triggered the wake signal, the device itself must supply this information and the S*x* andS0 disarm-wake callbacks should read it from the device.

#### Power Policy S*x* Wake Settings

After the driver creates its DO, it can initialize the S*x* wake settings:

* A UMDF driver calls IWDFDevice2::AssignSxWakeSettings.
* A KMDF driver fills in the WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS structure and then calls **WdfDeviceAssignSxWakeSettings**.

Table 6 lists the S*x* wake settings.

Table 6. Device S*x* Wake Settings

| Setting | Description | Possible values |
| --- | --- | --- |
| **Enabled** | Whether the device can wake the system from a low-power state. | **WdfTrue WdfFalse** **WdfDefault** (enabled unless explicitly disabled by a user with administrator privileges) |
| **DxState** | The device power state to which the framework transitions the device when the system enters a wakeable low-power state. | **PowerDeviceD1 PowerDeviceD2 PowerDeviceD3** |
| **UserControlOfWakeSettings** | Whether the framework provides a property page in Device Manager to allow administrators to control the device’s ability to wake the system. | **WakeDoNotAllowUserControl WakeAllowUserControl** |

The **DxState** value specifies the device power state into which the framework should put the device when it is armed for wake from S*x*. By default, the framework uses the value that is supplied in the device power capabilities.

**UserControlOfWakeSettings** indicates whether appropriately privileged users can control the wake policy of the device. If the value of this setting is **WakeAllowUserControl**, the framework automatically creates a Device Manager property page for the driver that allows a user with administrator privileges to enable or disable device wake. If both wake and idle are supported by the device and both allow user control of their policies, the wake and idle options appear together on a single Device Manager property page for the device, by default. The property page modifies the **IdleInWorkingState** and **WakeFromSleepState** registry values, which are stored in the **Parameters\Wdf** subkey for the devnode. Users and drivers must not access these registry values directly.

The driver can disable user control of wake policy by setting this value to **WakeDoNotAllowUserControl**. Most drivers, however, should allow users to control wake policy because some hardware configurations support wake poorly.

To assign the wake settings, a UMDF driver calls **IWDFDevice2::AssignSxWakeSettings** and a KMDF driver calls **WdfDeviceAssignSxWakeSettings**.

### How to Implement Wake from S0

Implementing support for device wake from S0 is similar to implementing support for wake from S*x*. Initialization typically occurs in the driver’s add-device callback. However, because support for wake from S0 is related to device idle support, the driver must implement idle support and indicate that it supports wake from S0 when it enables idle support, as described in “Idle Settings and Management,” earlier in this paper.

A driver implements S0 wake callbacks in much the same way as S*x* wake callbacks, typically during add-device processing:

* A UMDF driver implements the **IPowerPolicyCallbackWakeFromS0** interface on the device callback object.
* A KMDF driver implements power policy event callbacks as required, fills pointers to these callbacks into the WDF\_POWER\_POLICY\_EVENT\_CALLBACKS structure, and calls **WdfDeviceInitSetPowerPolicyEventCallbacks** to register the callbacks in the WDFDEVICE\_INIT structure.

Table 7 summarizes the power policy event callbacks for S0 wake.

Table 7. Power Policy Event Callbacks for S0 Wake

|  |  |  |
| --- | --- | --- |
| Event | UMDF callback | KMDF callback |
| Enable device’s ability to wake system from S0 state | **IPowerPolicyCallbackWakeFromS0:: OnArmWakeFromS0** | *EvtDeviceArmWakeFromS0* |
| Disable device’s ability to wake system from S0 state | **IPowerPolicyCallbackWakeFromS0:: OnDisarmWakeFromS0** | *EvtDeviceDisarmWakeFromS0* |
| Respond to device wake signal in S0 state | **IPowerPolicyCallbackWakeFromS0:: OnWakeFromS0Triggered** | *EvtDeviceWakeFromS0Triggered* |

The S0 arm-wake and S0 disarm-wake callback functions perform device-specific actions that arm and disarm the device to wake when the system is in S0, such as waking a network adapter when the user plugs in a cable. These functions are required only if the driver and device require such actions:

* If the device requires the same actions both to arm the device for wake from S0 and to arm the device for wake from S*x*, a KMDF driver can specify the same callback function in both the *EvtDeviceArmWakeFromS0* and *EvtDeviceArmWakeFromSx* fields of the WDF\_POWER\_POLICY\_EVENT\_CALLBACKS structure.
* If no device-specific actions are required to prepare the device to wake the system from S0, for example, a KMDF driver can omit the *EvtDeviceArmWakeFromS0* callback function and a UMDF driver can supply a stub. This is likewise true for the S0 disarm-wake callbacks.

The S0 wake-triggered callback, like the *Sx* wake-triggered callback*,* is an informational callback and certain caveats apply, as the previous section points out.

If a KMDF driver supports both wake from S0 and wake from S*x*, it fills in WDF\_POWER\_POLICY\_EVENT\_CALLBACKS with the necessary callbacks to support both wake from S0 and wake from S*x* before it calls **WdfDeviceInitSetPowerPolicyEventCallbacks**.

A driver indicates that a device supports S0 wake by specifying **IdleCanWakeFromS0**—or **IdleUsbSelectiveSuspend** for a USB device—in its S0 idle settings. For details, see “Device Power-Down Idle Support” earlier in this paper.

For a UMDF example that implements device idle and wake, see “Selective Suspend in USB Drivers” on the WHDC Web site.

## KMDF Example: Support for Device Idle and Wake

This example continues the hardware function driver in “KMDF Example: Plug and Play and Power Code in a Simple Hardware Function Driver” earlier in this paper. You can see the code for the driver’s *EvtDriverDeviceAdd* callback in Listing 7 in that section.

The Osrusbfx2 driver adds support for device idle and USB selective suspend by initializing the WDF\_DEVICE\_POWER\_POLICY\_IDLE\_SETTINGS structure and calling **WdfDeviceAssignS0IdleSettings**. Before the driver can initialize idle and wake support, it must determine whether the device supports these features. To find out, the driver interrogates the device from its *EvtDevicePrepareHardware* callback, which is shown in Listing 10. This function appears in the Driver.c source file.

Listing 10. Sample USB driver’s *EvtDevicePrepareHardware* callback

NTSTATUS OsrFxEvtDevicePrepareHardware(

IN WDFDEVICE Device,

IN WDFCMRESLIST ResourceList,

IN WDFCMRESLIST ResourceListTranslated

)

{

NTSTATUS status;

PDEVICE\_CONTEXT pDeviceContext;

WDF\_USB\_DEVICE\_INFORMATION deviceInfo;

ULONG waitWakeEnable;

UNREFERENCED\_PARAMETER(ResourceList);

UNREFERENCED\_PARAMETER(ResourceListTranslated);

pDeviceContext = GetDeviceContext (Device);

// Create a USB device handle to communicate with the

// underlying USB stack.

**status = WdfUsbTargetDeviceCreate (Device,**

**WDF\_NO\_OBJECT\_ATTRIBUTES,**

**&pDeviceContext->UsbDevice);**

if (!NT\_SUCCESS(status)) {

return status;

}

status = SelectInterfaces(Device);

if (!NT\_SUCCESS(status)) {

return status;

}

// Retrieve USBD information

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

**status =**

**WdfUsbTargetDeviceRetrieveInformation (**

**pDeviceContext->UsbDevice,**

**&deviceInfo);**

**waitWakeEnable = deviceInfo.Traits**

**& WDF\_USB\_DEVICE\_TRAIT\_REMOTE\_WAKE\_CAPABLE;**

// Enable wake and idle timeout if the device supports it.

**if(waitWakeEnable){**

**status = OsrFxSetPowerPolicy(Device);**

**if (!NT\_SUCCESS (status)) {**

**return status;**

**}**

}

status =

OsrFxConfigContReaderForInterruptEndPoint (pDeviceContext);

return status;

}

Listing 10 shows how a driver might determine whether its device supports wake. In this sample, the driver creates a USB target DO and then calls **WdfUsbTargetDeviceRetrieveInformation** to get the capabilities of the device and the underlying port driver. These capabilities are returned as a set of bit flags in the **Traits** field of a WDF\_USB\_DEVICE\_INFORMATION structure. The driver tests the value of the WDF\_USB\_DEVICE\_TRAIT\_REMOTE\_WAKE\_CAPABLE bit and, if it is true, calls the OsrFxSetPowerPolicy internal helper function to enable idle and wake support.

Listing 11 shows the code for OsrFxSetPowerPolicy. This function also appears in Device.c.

Listing 11. Initializing wake and idle support in a KMDF USB driver

NTSTATUS OsrFxSetPowerPolicy(

IN WDFDEVICE Device

)

{

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

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

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)) {

return status;

}

// Initialize the wait-wake policy structure.

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

**status = WdfDeviceAssignSxWakeSettings(Device, &wakeSettings);**

if (!NT\_SUCCESS(status)) {

return status;

}

return status;

}

The OsrFxSetPowerPolicy function is called from the driver’s *EvtDevicePrepareHardware* callback to enable idle and wake for the USB device.

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.

For USB devices that support selective suspend, the underlying bus driver prepares the device hardware to wake. Consequently, the function driver should supply an *EvtDeviceArmWakeFromS0* callback only if additional device-specific programming is required. The framework sends a selective suspend request to the USB bus driver when the idle time-out expires.

## Framework Actions Supporting Device Idle

The framework counts the I/O activity on all power-managed queues that each DO owns. If the driver supports idle for the DO, the framework starts a timer whenever the I/O count reaches zero. The timer is set to expire at the number of milliseconds that were specified in the **IdleTimeout** field that was most recently passed to **WdfDeviceAssignS0IdleSettings**.

If an I/O request arrives at a power-managed queue that belongs to the device or if the driver calls **WdfDeviceStopIdle** before the **IdleTimeout** period expires, the framework cancels the timer.

If instead the timer expires, the framework takes the required steps to transition the device out of D0 and into the device power state that the driver specified in the **DxState** field that was most recently passed to **WdfDeviceAssignS0IdleSettings**.

Regardless of the reason for the transition, the framework always handles the transition out of the D0 state in the same way. Thus, when a device transitions from D0 to D*x*, the framework invokes the driver’s callback functions according to the power-down sequences that were described in “Device Power-Down and Removal” earlier in this paper.

If the device is idling in its low-power state, the framework automatically returns the device to D0 whenever the count of I/O activity on any of the device’s power-managed queues becomes nonzero or when the driver calls **WdfDeviceStopIdle**. Again, the transition to D0 is always handled according to the power-up sequences that were described in “Device Enumeration and Startup” earlier in this paper.

## Framework Actions Supporting Device Wake

If a driver supports wake for its device, the framework calls the driver’s *EvtDeviceArmWakeFromSx* callback during a system transition to a lower power state other than S5 (the fully-off state), if the driver supports wake from the new system state. For example, assume that the system is transitioning to S3. If the driver supports wake from S3, the framework calls the driver’s *EvtDeviceArmWakeFromSx* callback. However, if the driver supports wake only from S1, the framework does not call the driver to arm the wake signal.

The following is the prototype for the *EvtDeviceArmWakeFromSx* callback function:

NTSTATUS EvtDeviceArmWakeFromSx(WDFDEVICE Device)

The framework calls this function before taking any action to transition the device to its D*x* state, such as calling *EvtDeviceD0Exit*. Within the *EvtDeviceArmWakeFromSx* function, the driver arms the device for wake from S*x*. If the driver cannot successfully arm the device, the callback returns an error and the framework continues the transition to the D*x* state without the device being armed for wake. If the system and device are later powered up again and then put to sleep, the framework by default again calls the driver’s *EvtDeviceArmWakeFromSx* callback. That is, the framework does not “remember” that the driver failed to arm the device during a previous power-down.

The driver can disable further requests to arm the device for wake from S*x* by returning the WDFSTATUS\_ARM\_FAILED\_DO\_NOT\_RETRY status from the *EvtDeviceArmWakeFromSx* callback.

If the device triggers the system to wake, the framework calls the driver’s *EvtDeviceWakeFromSxTriggered* callback. Because the framework handles all the work that is necessary to wake the system, this callback is strictly informative for the driver.

When the system returns to S0, the framework calls the driver’s *EvtDeviceDisarmWakeFromSx* function after the device returns to operation, that is, after the framework calls *EvtDeviceD0Entry*. The *EvtDeviceDisarmWakeFromSx* callback disarms the device for wake and reverses any other device-specific actions that were taken in its *EvtDeviceArmWakeFromSx* function.

Supporting wake from S0 is almost the same as supporting wake from S*x*. The only difference is that the framework invokes the driver’s *EvtDeviceArmWakeFromS0* and *EvtDeviceDisarmWakeFromS0* event callbacks when the device is ready to transition to or from the idle state, respectively. As with *EvtDeviceArmWakeFromSx* and *EvtDeviceDisarmWakeFromSx*, these callbacks are invoked before other driver callbacks that involve leaving or entering D0.

# Resources

Windows Driver Foundation (WDF) on the WHDC Web site  
<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

Plug and Play—Architecture and Driver Support on the WHDC Web site  
<http://www.microsoft.com/whdc/system/pnppwr/pnp/default.mspx>

Windows Driver Kit (WDK)  
<http://www.microsoft.com/whdc/Devtools/wdk/default.mspx>

Debugging Tools for Windows—Overview  
<http://www.microsoft.com/whdc/devtools/debugging/default.mspx>

#### Books

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

#### White Papers

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

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

Selective Suspend in USB Drivers  
<http://www.microsoft.com/whdc/driver/wdf/USB_select-susp.mspx>

#### Training Videos

Debugging Kernel-Mode Driver Framework Drivers

<http://www.microsoft.com/whdc/devtools/debugging/kmdf.mspx>

[Debugging User-Mode Driver Framework Drivers Training Sessions](http://www.microsoft.com/whdc/devtools/debugging/umdftraining.mspx)

<http://www.microsoft.com/whdc/devtools/debugging/umdftraining.mspx>

#### Samples in the WDK

Fx2\_Driver

%wdk%\Src\Usb\Osrusbfx2\Umdf\Fx2\_Driver

Osrusbfx2

%wdk%\Src\Usb\Kmdf\sys\final

Toaster Filter

%wdk%\Src\General\Toaster\Kmdf\Filter\Generic

USB Filter

%wdk%\Src\Usb\Osrusbfx2\Umdf\Filter

#### 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>